AD-A013 582

A MODEL OF HUMAN COGNITIVE BEHAVIOR IN WRITING CODE FOR COMPUTER PROGRAMS. VOLUME I

Ruven Brooks

Carnegie-Mellon University

Prepared for:

Air Force Office of Scientific Research
Defense Advanced Research Projects Agency

May 1975

A Model of Human Cognitive Behavior in Writing Code for Computer Programs

Ruven Brooks

May 1975

Vol. I.

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR - 75 - 1084	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A MODEL OF HUMAN COGNITIVE BEHAVIOR IN WRITING CODE FOR COMPUTER PROGRAMS, VOL I		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ruven Brooks		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D A0-2466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE May, 1975
		13. NUMBER OF PAGES 154
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research/NM 1400 Wilson Blvd Arlington, VA 22209		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A theory of human cognitive processes in writing code for computer programs is presented. It views behavior in terms of three processes, understanding, planning, and coding. The first of these consists of acquisition of information from the problem instructions and directions. This is used by the planning process to create a solution plan stated as a set of functional specifications in a language which is independent of the syntax of the particular programming language. The coding process converts this plan to code using a process named "symbolic execution" in which pieces of code

Block 20/Abstract

are assigned effects expressed in terms of the functions the programmer intends the code to perform in achieving the purpose of the program.

Within the framework of this theory, a more explicit model of the coding process was developed. The model is based on a production system and has been implemented as a computer program. Given plans taken from protocols of a programmer writing a series of short FORTRAN programs, it is able to generate the same code in the same order as the programmer did.

The model makes three assertions about programmer behavior in writing programs:

1. Programmers have alarge amount of specific knowledge about how to encode particular plan elements.

2. Programmers generate code by using the effects assigned to each piece to generate the next.

3. The basic units of a programmer's knowledge of language syntax are determined by the way in which he uses the language, rather than by properties of the syntax alone.

The implications of these assertions are discussed for the use of production systems to represent behavior, for teaching programming, for error analysis in debugging, and for the use of back-tracking in problem solving systems.

A Model of Human Cognitive Behavior in Writing Code for Computer Programs

Ruven Brooks

May 1975

Vol. I.

This work was supported in part by a grant from Xerox Corporation and in part by the Defense Advanced Research Projects Agency under contract no. F44620-73-C-0074 monitored by the Air Force Office of Scientific Research.

Abstract

A theory of human cognitive processes in writing code for computer programs is presented. It views behavior in terms of three processes, understanding, planning, and coding. The first of these consists of acquisition of information from the problem instructions and directions. This is used by the planning process to create a solution plan stated as a set of functional specifications in a language which is independent of the syntax of the particular programming language. The coding process converts this plan to code using a process named "symbolic execution" in which pieces of code are assigned effects expressed in terms of the functions the programmer intends the code to perform in achieving the purpose of the program.

Within the framework of this theory, a more explicit model of the coding process was developed. The model is based on a production system and has been implemented as a computer program. Given plans taken from protocols of a programmer writing a series of short FORTRAN programs, it is able to generate the same code in the same order as the programmer did.

The model makes three assertions about programmer behavior in writing programs:

1. Programmers have a large amount of specific knowledge about how to encode particular plan elements.

2. Programmers generate code by using the effects assigned to each piece to generate the next.

3. The basic units of a programmer's knowledge of language syntax are determined by the way in which he uses the language, rather than by properties of the syntax alone.

The implications of these assertions are discussed for the use of production systems to represent behavior, for teaching programming, for error analysis in de-bugging, and for the use of back-tracking in problem solving systems.

Table of Contents

1

A Theory of Computer Programming

1.1 Introduction

The development of large digital computers over the past 25 years has led to the development of theories of human behavior which view it in terms of information processing systems; since its inception in the mid-1950's the information processing framework for theories of human problem solving (Newell & Simon, 1972) has been applied to a large, and still growing, set of tasks which can be categorized along a few basic dimensions. One large group of tasks that has been useful in developing the mechanisms of the framework consists of laboratory tasks of short duration - a half hour or less - with relatively little direct applicability outside the laboratory. Subjects rarely have much experience with the tasks, and they do not attain any large degree of skill during the course of the study. Examples of such tasks are symbolic arithmetic ("cryptarithmetic") problems (Newell & Simon, 1972), the Moore-Anderson logic problems (Newell & Simon, 1972), and the problems, such as Tower of Hanoi and Missionaries and Cannibals, used with the General Problem Solver (Ernst & Newell, 1969).

Other work in the context of this framework has followed two major branches. One direction in which work within this framework has moved is towards studying certain simple, basic problem-solving processes which act as the building blocks for problem-solving in a wide variety of tasks. Included are studies of concept-identification (Simon & Kotovsky, 1963), children's and adult's seriation behavior (Young, 1973; Klahr, 1972), and children's conception of number (Klahr and Wallace, 1973).

A second major effcrt has been directed towards the study of behavior in more complex problems such as chess (Newell & Simon, 1972). Though the segments studied in the laboratory are usually short, chess differs from the tasks in the other groups in that it is a task that takes a long period of time; an entire game takes several hours. Additionally, and perhaps more important, behavior in chess is highly determined by skills acquired over a long period of time and which are developed to a diverse extent in different players.

One indication of this comes from a study (Chase & Simon, 1973) on differences in performance between a chess master and lesser players on tasks involving the reconstruction of a position on a chess board after viewing the position on another board. From analysis of timing data, the superior performance of the master player is attributable to his ability to organize his perceptions of the board in larger short-term memory chunks or units than those used by the other players, rather than by his having a larger short-term memory per se. In Simon & Gilmartin (1972), this ability was explained on the basis of the master's having available a larger number of patterns into which the pieces on the board could be organized. Since these patterns must be acquired from experience with and study of the game, these studies show clearly the importance of acquired skill, rather than inherent ability, in chess behavior.

The study of computer programming is an addition to this second group. Like chess, computer programming is a quite protracted problem-solving activity; some programs take as much as several hundred hours to write. Also like chess, programming involves skills which are built up over long periods of time; the existence of semester-long university courses in programming and of lengthy books written on programming techniques are an obvious basis for this assertion.

Programming differs from the other members of this second group in one major respect. In solving other problems in this group, the problem solver's behavior can be described in terms of a closed, small set of primitive knowledge elements and a closed, small set of operators which act on these knowledge elements. For example, a chess player's behavior in playing a game of chess can be described in terms of the player's knowledge of configurations of pieces and of a set of moves or move sequences which alter these configurations; while these may vary slightly between, say, openings and middle-game play, they remain essentially the same through out the game. These knowledge elements and the operators form together what is known as a problem space; the problem space together with the rules for when the operators should be invoked form a complete model of an individual's behavior. As the evidence presented later in this paper will show, the situation in programming is quite different; instead of solving the problem with a fixed set of operators and knowledge elements, new knowledge elements and operators are continuously being introduced as the problem-solver works towards a solution. This is a typical situation in a class of problems referred to as "design" problems. Another example of a design problem is the arrangement of fixtures in a bathroom (Eastman, 1969).

1.2 A Theory of Human Computer Programming Behavior

Although programming is an activity which is engaged in, in one form or another, by more than a million people (Boehm, 1972), research on human behavior in programming is very sparse. The earliest research along this lines is a small group of studies concerned with the effects of time-sharing versus batch on programmer productivity (Grant and Sackman, 1967; Smith, 1967; Schatzoff, Tsao, and Wiig, 1967; Gold, 1968). While changing economics have altered the answers to the basic cost questions that these studies were aimed at, an incidental finding, the extremely wide range of differences in performance across both problems and programmers, has been supported by later studies. (A recent study by Weinberg (1974) suggests that this may, in part, be product of programmer goals rather than a property of the task itself.) A second group of more recent studies has been in the area of debugging behavior (Rubey, 1968; Youngs, 1969; Gould and Drongowski, 1972; Gould 1974). While it is difficult to summarize the results of all these studies, it is notable that they all indicate that certain language structures appear to cause particular difficulty in debugging. Worthy of mention because of its title, the book, **Psychology of Programming** (Weinberg, 1971), is directed more at the social psychology of the programming environment than at actual programmer cognitive behavior. Finally, most recently there has been research done on the effect of certain language constructs on program understandability (Sime, Green and Guest, 1973; Weissman, 1973). It has elaborated the finding hinted at in the debugging studies, that some language constructs are easier or better to program with than others.

The research presented here takes a different approach than these previous studies. Instead of attempting to measure individual variables associated with programmer behavior, it presents a theory of the program writing process. The theory is based on a set of ideas developed by Allen Newell (1974) (i) and lies within the

----

(i) The elements of the Newell theory that are used here are: (1) Development of plans by heuristic search consisting of successive functional elaboration in which functional specifications invoke structures which, in turn, require further functions. (2) Generation of code by a symbolic execution process in which, first, code is laid down and then consequences are generated from it. This consequence generation produces a large number of sub-problems. (3) Solution of these sub-problems by a recognition framework of the information processing approach to human problem solving.

1.2.1 The Scope of the Theory

Though computer programming has been spoken of as though it were one task, in fact, a number of different tasks are included under this heading. They include, among many others, writing specifications for programs, writing programs given a set of specifications, debugging programs that another programmer has written, and writing documentation for programs. The particular task that has been selected for the focus of this theory is one in which the programmer is given a description of the input data and a specification of the desired output. The programmer must then find an algorithm, including the selection of a representation within the program for the external data, to produce the desired output and implement the algorithm in the programming language. As a working situation, it is one which occurs frequently in scientific and educational programming environments and as a sub-part of almost any larger programming task.

The theory is not yet formulated to include debugging behavior or situations in which different parts of the task are performed by different people, for example, as in large system writing projects. These restrictions are due mostly to limitations of resources, rather than to any inherent properties of the theory that are apparent. Some justification for the debugging ommission is that, since program writing proceeds program debugging chronologically and since the debugging process includes or makes assumptions about the processes which generated the code in the first place, development of a theory of program writing ought to take precedence. Similarly, a theory of how one individual performs the several parts to this task in sequence takes

----

process. Together with symbolic execution, this implies goal control dependent on the problem structure rather than via a goal stack.

precedence over theories about how the task is performed by several individuals interacting.

1.2.2 The Structure of the Theory

The theory consists of three basic processes, understanding, planning, and coding, though this work will focus only on coding. It asserts that each of these processes occurs one or more times in every programming task. The relationships among the processes, particularly regarding the way they invoke each other are described in the following sections.

1.2.3 Understanding

When a problem-solver is presented with a problem, he has a variety of sources of information, both internal and external, available about it; these include his general world knowledge, knowledge about the general type of problem at hand, reference works such as programming language manuals, and last but not least written or spoken problem directions. Before he can actually start work on the problem, he must use these information sources to build representations of the basic elements that the problem deals with and of their properties. Specifically, he must have representations of the initial state of the problem, the desired final state or goal, and one or more operations which he can apply, appropriately, to begin the transformation on the initial state. The process of building these representations is referred to as "understanding" in this theory; it is one of the three basic problem-solving processes which make up the theory.

The model for the understanding process that is adopted for this theory is based on one developed for a variant of the Towers of Hanoi task (Hayes & Simon, 1973). In their study subjects were given a single set of written instructions and had to solve the problem described in it. Their theory for the task proposes two basic processes, an UNDERSTANDING process and a SOLVING one; only the UNDERSTANDING process will be discussed here. In their theory, the UNDERSTANDING process extracts information from the problem description until enough information has been extracted to permit the SOLVING process to begin work. The SOLVING process then runs until either the problem is solved or the problem solver runs into difficulties; if this occurs control may then return to the UNDERSTANDING process. The UNDERSTANDING process consists of two sub-processes, a LANGUAGE interpreting process and a problem space CONSTRUCTION process, which alternate in the same manner as the UNDERSTANDING and SOLVING processes. Explicit mechanisms are presented for the internal structures of the LANGUAGE and CONSTRUCTION processes, but no mechanism is provided for how the alternation between the two takes place. Similarly, while the general structure is outlined, no specific mechanisms are provided for the alternation between UNDERSTANDING and SOLVING.

Not surprisingly, evidence of an understanding processes, in the form of alternation between reading directions and reasoning about what they say, is also seen in records of behavior in programming problems. In essential structure, this understanding process is presumed identical to the one described by Hayes and Simon; information extraction and incorporation phases alternate until understanding is complete. A major difference from their model occurs in the relationship between understanding and the other parts of the problem-solving process. When understanding terminates the outcome in their model is a problem space, a closed, small set of operators and knowledge primitives; the final solution to the problem takes place in this problem space. In programming, as mentioned, problem solution does not take place in the same sort of problem space as fo~ non-design tasks; what takes place instead when understanding terminates is the second of the three processes which make up the theory, planning.

1.2.4 Planning

Planning is the second of the three processes postulated by the theory. The type of plan produced by it can best be described as a method for solving the programming problem; it consists of specifications of the way in which information from the real world is to be represented within the program and of the operations to be performed on these representations in order to achieve the desired effects of the program. These methods are used as schemas or templates to guide the writing of the actual code in much the same way as plans are used to guide solutions of logic problems by the General Problem Solver (Newell and Simon, 1972).

It is an assertion of the theory that such a plan exists for nearly every programming problem that fits within the basic task definition. The basis for this assertion of the existence of plans is one of sufficiency. The space of possible programs is huge, even in comparison with other complex tasks; compare, for example, the number of possible, "reasonable," programs with the number of "reasonable" chess games. Search of even a small fraction of this space is not feasible; therefore, programming must involve extremely powerful heuristics which eliminate search or reduce it to a trivial level. In nearly all programming languages, each line of code actually involves a great many decisions - basic statement types, variable and expression choices, labels, etc. - and any heuristic which evaluates code on the basis of a single decision is not likely to have sufficient power. Only the use of plans is a powerful enough heuristic to make program writing feasible.

Plans are expressed in a functional language of the sort investigated by Newell and Freeman (1971); functions specified in the language invoke structures which, in turn, require other functions, a type of behavior which may be characteristic of the whole class of design problems. These functional specifications are organized in sequences in the order in which they are to be carried out. If the program is large and complex enough, these sequences may be nested in one another so that one sequence acts as a further elaboration or clarification for a step in another sequence. An example of this might be in a program to print all the odd numbers in an array. The plan could appear as:

1. Go through the array.
2. Test each number to see if it's odd.
3. If it is, print it out.

The second step might require the further elaboration:

2a. Represent each number as an integer.
2b. Divide it by two.
2c. Multiply it by two.
2d. Compare the result with the original number.
2e. If they are not equal, report that the number is odd.

It is worth comparing the use of this functional language for planning in programming with the content of plans in the planning version of GPS for the logic task (Newell and Simon, p.428; 1972). GPS plans by first solving the problem using a set of abstracted, usually simplified, operators to solve the problem and then using this solution as a plan to guide solution using the original set of operators. The way the latter is done is to convert each of the operators in the abstracted solution into an apply-type goal for solution in the original space. These apply goals specify a function to be invoked, and they often result in the creation of sub-goals for the production of certain structures. When used in this way, the languages of means-ends specification used in GPS is equivalent to the functional language used by human programmers in their plans for programs.

Also relevent is the language accepted by the Functional Description Compiler routine of the Heuristic Compiler (Simon, 1972). The input to this routine is a set of specifications of functions or operations to be performed. The language used to state these functions is independent of the syntax of the target language and at approximately the same level of abstraction as that of plans in this theory.

For plans to operate successfully as heuristics, the functions specified in them must be more global and more general than those available as primitives in the programming language. There are two reasons for this. The first is so that each function in the plan generates several statements in the program; if this were not the case, then there would be nothing to insure consistent use of variable names and the like, and to guarantee that a quantity calculated in one line would not be immediately recalculated in the following line because it was needed again.

The second reason is that there are some decisions about the structure of a program which carry across many lines or sections of code and which cannot be expressed in the primitives of most programming languages. One example of this kind of decision might be the selection of a linked list data structure in a FORTRAN program. The use of this particular data structure will be important within the generation of many lines of code, but there is no way to express this decision at the level of the primitives available in FORTRAN. This reasoning supports the idea that the functions specified in plans are more global and general than those available as primitives in the programming language. which carry across many lines or sectors.

An important question in discussing programming plans is to what extent these planning functions are sufficiently global and general to be used across different programming languages; this is equivalent to asking to what extent the plan that the programmer creates is dependent on the language in which the program will be written.

One possibility is that, for a given problem, an entirely different plan will be created for each programming language in which the program is written. There are two substantial arguments against this position. One is the introspective report of many programmers that they are able to think about the solution to a programming problem in terms independent of the syntax or semantics of a particular programming language. A second is the informal finding that, when given a problem description and several, similar, languages in which the program might be written, most "multi-lingual" programmers have no difficulty in creating plans that will work in any of the languages.

A second possibility, representing the opposite extreme, is that a single planning process and functional language serve for all the languages that a programmer knows and that the plan is dependent on only the problem alone. This would imply that, for a given problem, the programmer would use the same plan whether the target language were LISP or FORTRAN. In fact, programmers do not usually use plans involving lists if the program is to be in FORTRAN nor do they select algorithms using arrays for LISP programs, indicating that the creation of a plan is to some extent dependent on the programming language.

The locus of this language dependency seems to be shared between two attributes of programming languages. One is in the data structures (and access methods) available in the programming languages; languages which share common data structures probably also share common plans and a common planning process. This would be consistent both with the observation that programmers are able to create common plans for pairs of languages such as ALGOL and FORTRAN, which both have array data structures, and with their creation of different plans for LISP and FORTRAN, which do not share a common type of data structure.

The second attribute which determines whether different languages share common plans could be control structure. Even two languages sharing common data structures may not be able to use the same plans if one has a primarily recursive control strucure while the other is strongly sequential. Consider the differences in the flow-charts for a LISP recursive factorial program versus a FORTRAN iterative one. This may also play a roll in the report of many programmers that they are unable to create common FORTRAN-LISP plans, even when the LISP has an array features.

Planning does not take place as a single operation; instead, a process of step-wise refinement takes place in which each step makes part of the plan more detailed. The terminating condition for the refinement is that some (reasonably large) part of the plan is sufficiently detailed so that the programmer feels that he knows how to begin to translate it into code, even though all of the details of the code are still unknown. At that point the final process in the writing of programs, coding, takes over. The coding process operates on a piece or part of a plan until either code is produced or some criterion is met which causes the coding process to report failure; when failure occurs information is passed back to the planning process which again attempts to produce a codeable plan. This alternation is very much like that which occurs between understanding and solving in the Hayes and Simon model.

As has been mentioned, the primary focus of this will work will be on the coding process. Since the coding process is dependent on the outcome of the planning process, a brief exploration of possible ways in which plans could be generated is worthwhile.

To begin with, a recognition process must play a large role. In many, if not most, programming problems, planning takes place fairly rapidly; in the problems studied in this research, it began less than 1.7 minutes after the subject had received the problem description and lasted less than 5 minutes in a 25 minute problem. Little or no evidence of of any kind of search activity was seen. This suggests that what takes place is basically a match between characteristics of the current problem and stored information about similar problems that have been done in the past. The information retrieved this way could be used either to retrieve a stored plan directly or to provide other information which could guide the operation of a simple, fast-acting plan assembler. This recognition mechanism also implies the existence of mechanisms for extracting characteristics from current problems and mechanisms for abstracting plans from solved problems. Building a theory of the latter will be an especially challenging problem, as other work on plan abstraction has demonstrated (Fikes, Hart, and Nilsson, 1971).

While the recognition system may take care of the overwhelming majority of cases, other mechanisms will be necessary for the cases in which stored information is not sufficient. These might be divided into two broad, general classes: those which use programming knowledge and those which use knowledge from the real-world problem domain for which the program is being written. In the former are included patching and rearranging existing plans (Sussman, 1973); generalizing from examples; and the use of diagrams or drawings (Gelernter, Hansen, and Loveland, 1960). In the latter are included all those situations in which the programmer goes outside the programming domain and uses knowledge about the intended use of the program, relationships among the data, etc. to solve the problem; an example might be use of knowledge about a company's accounting policies to come up with a plan for writing a payroll program.[i]

1.2.5 Coding

The third of the three processes in the theory is coding. For human programmers, the basic cycle for the generation of code consists of using the plan to select and write a piece of code, assigning an effect or consequence to the code that has been written, and comparing the effect or consequence to the stipulations of the plan. The results of this comparison are used to select and write more code or to change the code that has been written; in turn, an effect is assigned to this new code which is compared to the plan. This cycle continues until the cumulative effect of the code meets the requirements of the plan or until some condition, such as effort expenditure, is met which indicates that the piece of plan is not codeable.

A similar cycle occurs if the programmer is going over code he has already written, as often occurs when checking over code or going back to add initializations. In this case, the programmer takes each piece of the written code, assigns an effect or consequence to it, and compares it with either the plan or with the effect he assigned to the code when he wrote it the first time.

The effects that are assigned to code are based on the differentiations among the data that the program must actually make in order to accomplish its purpose. Consider as an example a program for printing all the odd numbers in a set of integers. The program must differentiate between odd and even numbers in order to perform this task. An effect that could be assigned to a line of code in this program might be "if the

---

[i] It is conjectured that one characteristic behavior in the writing of large, as versus small, programs or systems is that more of the problem-solving behavior involves the use of this non-programming knowledge.

number is odd, this branches to statement 50," a statement which uses the information about the odd-even distinction. As more lines of code are written, their effects are accumulated in a manner which also makes use of these differentiations; thus, the effects, "this loops through all the numbers" and "if the number is odd, branch to statement 50," might be combined to give "this tests each number to see if it's odd." When effects of this type are assigned to whole segments of code, the result is that the code is executed with symbols such as "odd number" replacing the real data. Hence, the whole process has been named "symbolic execution."

The language in which these effects are expressed is a functional one that resembles closely the language used to express plans. There is, however, an important distinction between effects in coding and plan statements in the coding process. Consider the plan statement, "loop through the list," and the effect statement, "this loops through the list." The first is the statement of an intention, not an assertion of a result that has been achieved; as such, no attempt is made to check its accuracy. The effect,[i] on the other hand, is a statement about what the programmer believes code actually does. Even without actually having the machine execute the code, the programmer can double-check this statement by making a second pass through the code himself. In protocols of programming behavior, this distinction between planning

----

[i] If the plan is a very complex one, it may happen that the programmer attempts to verify, while seemingly still in the middle of the planning process, that some part of a plan does indeed achieve an effect that some other part of the plan requires. If the reasonable assumption is made that what actually takes place is a brief coding episode in the middle of planning, this kind of behavior is also consistent with the intent-effect distinction.

statements of intent and coding statements of effect appears clearly enough so that it may be used to identify which of the two processes is taking place at a given point.

An example of a complete symbolic execution cycle for the problem just mentioned might start with the plan element, "test each number to see if it's odd." For a FORTRAN program, the programmer would begin by writing DO 10 I=1,100 and assigning it the effect, "this loops through all the numbers." Given this effect and the plan, the programmer might next write IF(L(I)/2*2 .NE. L(I)) GO TO 20 and then assign it the effect, "this tests whether its odd and goes to 20 if it is." Finally, after closing the DO loop by writing, 10 CONTINUE, the programmer would summarize the effect of all three lines as "this loops through all the numbers and tests each one to see if its odd." Since this matches the plan element, program writing would proceed to the next plan element.

An alternative possibility in this example illustrates another aspect of symbolic execution. Suppose the programmer had known only a test for even parity. Since it was the only parity test available, he might have written IF(L(I)/2*2 .EQ. L(I)) and assigned to it the effect, "this tests whether its even." Noting that the plan requires the opposite effect, he would then alter ".EQ." to ".NE." to obtain the test for odd parity. The general principle illustrated here is that, confronted with erroneous code, the symbolic execution process attempts to patch or modify it to obtain the desired effect.

This patching or modifying behavior is one of the main characteristics that distinguish symbolic execution from the sort of goal tree building and backtracking behavior seen historically in programs such as the Logic Theorist (Newell and Simon, 1972) and, more recently, in systems such as PLANNER and CONNIVER (Hewitt, 1971; Sussman & McDermott, 1972) These systems rely heavily for problem solving power on the ability to backtrack to a previous, successful position. In symbolic execution, on the other hand, attempting to modify or add on to what had already been done takes precedence, and backtracking is an infrequent event.

When backtracking does take place in coding, it may occur at several levels. In addition to attempting to code the plan element in an alternative way, it may be decided that it is the plan which is at fault. When this happens, a return is made to the planning process, and an attempt is made to find a plan or piece of plan which is easier to code. This new attempt in planning may even require a return to the understanding process to re-interpret the problem. If the understanding process is considered to be a "top" level process and the coding process a "bottom" level one, then this ability to backtrack to the planning and understanding process represents a "bottom-up" process, and both bottom-up and top-down processes take place in programming.

1.3 The Design Task Nature of Programming

In relating programming to other tasks which have been studied within the theory of human problem solving, it was stated that programming differed from other tasks of similar proportions in that new knowledge elements and operators were introduced continuously during the course of problem-solving. This design task nature of programming can be derived by two independent lines of reasoning based on two different aspects of the processes described in this theory. First, the recursive interaction among the three processes makes it impossible to represent the entire task in a single problem space. Even if each of the three processes can be represented compactly as a problem space, the union of these spaces would contain too many different operators and knowledge elements to fit within the problem space definition.

A second argument for the design-task nature of programming is that both the planning and code generation processes have characteristics which make their representation as problem spaces extremely unlikely. In the previous discussion of planning, two general classes of methods for the construction of plans were presented. The first class of methods involved rapid, recognition-like processes. Behavior produced by these methods might possibly be fit into a problem-space characterization since they could involve classifying problems using only a few basic elements and then generating the plans using a small set of operations. The fit of this characterization would, however, probably not be a very good one, since some of elements and operators would only be used once in given situation. For the second set of methods, the situation is even worse. Use of diagrams, playing with examples, etc. are behaviors which would be very difficult to handle using a closed set of operations and knowledge elements. For methods involving use of knowledge from other domains, in particular, a single, small set of operations and knowledge elements will be plainly unsatisfactory. Since most of the methods which make up the planning process cannot themselves be represented in problem spaces, it is clearly impossible to represent the entire process this way.

The unsuitability of problem-space representations for the code generation process is derivable from the way in which sequences of code are created. As each piece of code is laid down, the effects or consequences that are assigned to it serve as part of the invoking conditions for laying down of the next piece. An alternative way to express the same process is to consider the plan element and the effect of a previous pieces of code as together constituting a sub-problem for the creation of the next piece. A recognition process generates the required code to solve the sub-problem, and the effects assigned to this new code serve as part of a new sub-problem. From this perspective, code generation involves the statement and immediate solution by recognition of a huge number of sub-problems. Representation of these problems and their solution in terms of a small, closed set of operators and knowledge elements is impossible for most coding behavior, ruling out the use of problem spaces to model coding behavior. Thus, since both the planning and coding processes cannot be represented as problem spaces, an overall problem-space representation for the programming task is not possible.

1.4 Overview and Forward

The theoretical framework presented in this section is a complex one; it specifies three major processes, each of which may have a unique internal structure, which can interact extensively with each other. In order to verify this framework against experimental data, it would be necessary to completely specify the internal structure of all three of the processes and to spell out precisely the way they interact. Doing this specification and verification for the complete framework would involve unavailable resources of time and experimental effort. Therefore, this work will focus on presenting a complete model and verification for only one of the three processes in the theory, coding.

2

A Data Base for Testing the Theory

In this section, the experimental situation and data used to develop and verify the model of coding will be presented. While a more conventional format would be to present this information after presentation of the model, presentation here has two advantages: it will make understanding of the model easier by giving readers an actual situation to refer to, and it permits some further explication with reference to an actual programming situation of a few aspects of the framework presented in the previous chapter. (Readers who find the conventional ordering more comfortable may read the next chapter before reading this one.)

The data on which this demonstration is based consists of behavioral records of a programmer writing and debugging 23 short programming problems. (Even though the theory does not include debugging behavior, the subject was asked to debug the programs, both to collect the data for possible future use and, more important, to insure that they were under the same sorts of constraints in writing the programs that they would be under outside the experimental situation.) The 23 problems were created both to be similar enough so that the same behaviors would be repeated in different problems and to differ enough to minimize learning effects across problems. Additionally, they each had to be short enought to be written and debugged in a 2 hour period.

All the problems involve manipulations on an array, L, 100 in length, filled with random integers. For the first subject for the first 8 problems the numbers lay in the range, -10000 to +10000; this proved somewhat difficult to read in print-outs so that the range was changed to -1000, +1000 for the remaining data collection. A second array, M, also 100 long, was provided to be used for indicating certain information about L. Each problem was given a proper name (e.g. LARRY) to avoid any inference of ordering among them. The problems are listed in Appendix 1.

Subject instructions were to write and debug each of the programs in FORTRAN and to talk aloud about what was being done. Since FORTRAN as a programming language is currently in disfavor with much of the computer science community, a few words of explanation for its choice in this study are in order. FORTRAN was selected because it is probably the most widely "understood" programming language, and its use guarantees understandability of this study by readers unfamiliar with PASCAL, ALGOL, LISP or a host of other languages whose use may lead to superior programming practice. Use of one of these other languages would certainly have lead to different programmer behavior than that observed with FORTRAN, if for no other reason than that different languages have different syntax; however, there is, as yet, no reason to believe that this different behavior would require different mechanisms than those already in the theory.

While working on these problems the programmer could use both paper and pencil and a 10 character-per-second, hard-copy computer terminal connected to an interactive computer system with which the programmer was familiar; alternation between the two could be made as desired. His behavior was recorded via a throat microphone and a video tape recorder with the camera placed behind the subject. These recordings plus the written materials and the computer terminal output provide the basic data on which this study is based.

For each program, the programmer was given a printed description of the problem to be programmed and a copy of the general instructions. Whenever he was ready to enter the program into the computer he was given the name of a file on the computer system containing the necessary instructions to read the data into to the L array, set the M array to zeroes, and write out the L and M arrays at the end of the program.

The subject employed in this study was a very experienced programmer. At the time of this study he was a graduate student in computer science at Carnegie-Mellon University. Prior to serving in this study, he had more than 10 years of programming experience. This included writing LISP interpreters for the Control Data G20 and Univac 1108 computers, writing an assembly system for the Univac 1108 in FORTRAN, and work on the 1108 ALGOL compiler. He has written substantial programs in FORTRAN, ALGOL, LISP, SNOBOL, IPL-V, APL, and BASIC as well as several other, less-known languages and dialects. He had been employed as a programmer by Carnegie-Mellon University, International Business Machines, and the National Bureau of Standards. Finally, from fall of 1969 through the fall of 1972 he taught introductory programming courses at Carnegie-Mellon University.

The programmer was paid $2.50 per hour for his time and worked in sessions of up to 2 hours in length, the length of any given session being determined by the programmer. With the exception of the 15th problem, problem KEVIN, each problem was completed in a single session.

To those not familiar with the information-processing approach to problem-solving, use of only a single subject may require a few words of explanation. The information-processing theory of problem solving regards behavior as highly history dependent; any single piece of behavior can be understood only in terms of it own particular precedents. For programming in particular, averaging across the behavior of severr individuals would obscure these precedents. An appropriate research design for studying programming must, therefore, be based on examination of extended sequences of behavior. While, ideally, this should be done for a wide range of individuals and situations, constraints on resources have limited this study to a single individual.

2..1 Preliminary Processing

The video tapes were prepared for analysis by transcribing them into written protocols. The transcription was performed by a single listener (the experimenter) and was done in two major passes. The first pass transcribed the spoken information only; lines from this pass in the protocols in Appendix 1 are preceeded by an "S." The second pass was used to extract information from the visual record, such as writing behavior, etc; lines from it are prefixed with an "A."

The breaking of the spoken information into lines in the transcription was made according to two rules: First, a break was made whenever the subject paused, even if the pause was in the middle of a word or phrase. Second, if the speech was relatively continuous, breaks were made between major clauses. The segmentation into lines in the protocol is, thus, a rough indicator of low-level behavioral units in problem solving.

2..2 Times

Timing information was obtained using the digital counter on the video tape recorder; the times are accurate to .3% (an absolute error of 5 seconds in a 25 minute protocol). They are given in Table

2..2.1. In addition to total time, separate figures are given for writing and debugging time. Writing time was defined to be the length of time from receiving the problem description until the program was executed or compiled for the first time. (Another possible alternative, time until the subject began typing in the program, was not used, since the subject occasionally began typing in the program before he had completed writing it.) debugging time was defined as the time from first execution until the subject asserted the program was operating correctly. The problems took a mean of 25.9 minutes to write and 15.0 minutes to debug for a total problem solving time of 40.9 minutes.

This is equivalent to an average of 4.9 seconds per line of protocol for program writing.

Table 2..2.1
Writing and Debugging Times for the 23 Problems
(in minutes)

Problem	Writing	Debugging	Total	Ratio	Lines
HENRY•	19.2	11.1	30.3	1.7	27
DAVID	55.8	4.4	60.2	12.7	43
WILLIAM	25.0	4.8	29.8	5.2	20
JOHN	27.5	10.2	37.7	2.7	12
PETER	6.4	8.7	15.1	0.7	9
CARL	13.1	5.3	18.4	2.5	9
BRIAN	43.7	11.3	54.9	3.9	23
PAUL	39.1	22.2	61.3	1.8	26
STEVEN	18.1	3.9	22.0	4.6	25
RALPH	22.9	25.6	48.5	0.9	17
RICHARD	13.3	3.0	16.2	4.5	10
ROBERT	22.2	12.7	34.9	1.7	26
HAROLD	42.3	55.4	97.6	0.8	35
DONALD	35.7	10.3	46.1	3.5	26
KEVIN	66.0	73.9	139.9	0.9	74
GERALD	60.4	36.8	97.2	1.6	65
FRANK	9.8	4.4	14.2	2.2	13
LARRY	4.4	4.9	9.2	0.9	10
LEE	24.0	17.5	41.5	1.4	30
PHILLIP	17.2	12.7	29.8	1.4	21
SAM	5.2	0.8	6.0	6.9	9
ALLAN	3.8	1.6	5.4	2.3	10
OSCAR	20.4	2.9	23.3	7.1	21
Means:	25.9	15.0	40.9	1.7	24.4

Several things are noteworthy about these times. The first is the wide range of values for writing and debugging times. Extreme values differ by almost an order of magnitude. Since the problems were designed to vary moderately in degree of difficulty, it is of interest whether the observed differences in times are a product of problem difficulty or whether they have some other source.

One possible measure of actual problem difficulty is the number of lines of code required to do the problem. It is an imperfect measure since both inefficient solutions and certain unusual problem characteristics may lead to inflated values; however, for this problem set and programmer, the measure is probably a useable one. It correlates .69 and .75 with writing and debugging time respectively. While these are substantial and indicate a strong relationship between problem difficulty and writing and debugging times, they still leave a considerable portion of variance unaccounted for. This missing variance, of course, represents differences in program writing difficulty that are disproportionate to the number of lines in the program.

A second noteworthy point about the times is the correlation of .69 between debugging and writing times. This is, again, a substantial correlation; it indicates that problems which take a long time to write also take a long time to debug, a conclusion also made in other studies (Youngs, 1970). As in the previous case, the correlation is still considerably less than one, suggesting that there may be sources of difficulty in debugging which are independent of difficulty in problem solving.

Finally, there is a high ratio of writing to debugging time. The median ratio is 1.7 to 1 and only 5 of the 23 ratios are less than 1 to 1. Other studies have found that debugging time almost invariably exceeds writing time, occasionally by as much as 4 to 1 (Youngs, 1970; Rubey, 1968). It is difficult to pinpoint the cause of this difference, but possible explanations might include the high skill level of the subject, the nature of the problems, and the availability of "canned" code for doing input and output.

2..3 Verifying the Process Structure

Using the following set of criteria, the occurences of each of the three procesaes were identified in the protocols. (The complete set of classifications is shown in Appendix 3.)

Understanding

1. Reading the directions or problem statement.

2. Questions to the experimenter about problem interpretation.

Planning

1. Material, up to the writing of lines of code, which follows phrases such as "the way I would do this would be to ....." or "that seems similar to another problem I did" and which consists of a statment of a general solution to the problem, usually in terms which don't refer to a specific programming language.

2. References to knowledge domains outside that of programming, for example, an inquiry into the mathematical properties of prime numbers. While this sort of inquiry is usually part of the Understanding process, it may also take place in situations in Coding situations in which the programmer clearly already understands what is desired of him, but in which the additional knowledge from the outside domain is necessary to selecting a method.

3. Coding in a language other than the one in which the program will finally be written if no attempt is made to check or verify the effects of these lines of code.

Coding

1. Statements of an intent to generate code, such as "now I need a DO loop."

2. Statements of code being generated.

3. Statements of the effects of code that has been generated, particularly the assignment of hypothetical or symbolic values and the "execution" of the code for these values.

Item 3 under Coding requires an additional word of clarification. In protocols, PAUL and STEVEN, the subject first "solves" the problem in a pseudo-ALGOL. While it may seems as if this behavior ought to classified as Coding, it is, in fact, better classified as planning behavior. The code that is written this way consists only of outlines of main structures with few or no details and with many departures from ALGOL syntax into natural language statements. Additionally, phrases, such as "what I want to do is" followed by a statement of a line of code, indicate that what is being stated are the programmer's intentions, rather than effects that have already been achieved. For this reason, this coding in an alternate language is classified as part of the planning process (i) .

To clarify these rules, the following example from problem RALPH is presented. (This protocol was selected at random from those not used in other analyses; the program writing portion of it is given in Appendix 2.)

Problem RALPH

1:17

This segment consists of first reading the problem directions and then asking the experimenter for clarification. It is classified as Understanding.

18:96

Line 18 is "well, I'll do it the same way" and what follows up to line 28 is a statement of two possible alternative general solutions. From line 29

(i) A second argument that this behavior is planning, not coding, is simply that there is otherwise no justification for the extra effort required to write the program first in one language and then convert it to another language, particularly when the subject has had considerable experience programming in the final language to lines 96, the subject inquires into the properties of prime numbers. The whole segment is, therefore, classified as planning.

97:180

This whole segment consists of alternation among (1),(2) and (3) under Coding.

The following table gives some summary statistics about this classification for all 23 problems.

Table 2..3.1

Process	Occurs	Aver. Time (Secs.)	% Time
Understanding	1.39	101.4	6.9
Planning	1.91	313.4	21.2
Coding	1.87	1060.5	71.9

The Occurence column contains the mean number of times per problem the process in that column occurred in the 23 problems. The Aver. Time is the average total amount of time spent in that process in each problem. The % Time heading gives the percentage of the total program writing time spent in that process across all problems. Times were calculated by multiplying the number of lines by the time per line.

Note first that Planning and Coding occur about twice per problem where Understanding takes place a bit more than once. Note also that Coding accounts for a huge amount of the total time spent on a problem. Using this information, a good characterization of the problems in this study would be that the problems are easy to understand and the programmer can easily find a solution method for them. This method, however, requires considerable work to actually implement.

3

A Model of Coding Behavior

3.1 Introduction

As mentioned at the end of the first chapter, the focus of this work is on a model of the coding process to be applied to the behavior seen in the protocols. The model has been implemented as a computer program written in the University of California at Irvine dialect of the LISP programming language (Quam & Diffie, 1974; Bobrow, Burton & Lewis, 1973). The program runs on the Carnegie-Mellon University Computer Science Department Digital Equipment Corporation PDP-10 system. Except where reference is made to specific programming conventions, "model" and "program" may be considered synonymous for the rest of this discussion.

Since the model is intended to cover only coding behavior, the program operates as a theory of behavior only for segments of protocol which meet the criterion, discussed in the previous chapter, for being classified coding. For these segments, the program is a theory of behavior in two respects:

1. It generates code in the same order as does the subject in the protocols; in particular, it makes the same sort of corrections and modifications to code as he does.

2. The knowledge state of the program - the information the program contains about the status of the solution to the problem - changes in the same fashion as is seen in the protocols.

This chapter discusses the basic mechanisms and data structures selected for use in the model and discusses the psychological rationale for each selection. The suceeding chapter applies the model to some of the protocols; chapter 5 examines the psychological assertions that the model as a whole makes and presents additional evidence for them.

3.2 General Structure of the Model

The theory of human problem solving in which this theory of programming is embedded provides a framework for the structure of problem-solving models. This framework forms the basis for the structure of the model presented here; it includes two memories, each with unique accessing and storage characteristics, an overall control structure, and several elementary processes which serve as primitives for building larger-scale problem-solving activities. The following section is a discussion of this general structure.

The memory structures specified by the framework are a short-term memory (STM) and a long-term memory (LTM). The short-term memory (STM) has a fixed capacity of a small number (less than 20?) of chunks or symbols "each of which can designate an entire structure of arbitrary size and complexity in LTM" (Newell and Simon, 1972 ; p.795). Read-write time for STM is very short, perhaps on the order of a tenth of a second. During the course of normal problem-solving behavior, information rarely stays in STM more than a minute.

In contrast, long-term memory (LTM) is assumed to be very large or infinite in capacity. Write times are on the order of 5-10 seconds while read times on the order of a tenth of a second. No information is ever actually lost from LTM, but it may become inaccessible in varying situations for varying lengths of time.

Though they may not be needed at all in solving some types of problems, mention should be made of the system's ability to make use of external memories (EM's). EM's include such things as blackboards, paper-and-pencil, switch settings, or even the arrangement of physical objects, such as the use of paper cut-outs in a furniture arrangement problem. For the subject in this study, the paper on which he wrote his programs and the terminal print-out served as EM's. Read times were a function of the subject's reading speed and were probably on the order of 1-5 milliseconds per chunk (based on a reading speed of 200-1000 wpm. with each word a chunk). Write times depended on the subject's writing and typing speed. An approximate range would be 200 milli-seconds to 1 second per chunk, if a chunk is considered to be a five letter word and typing speeds are 60-300 wpm.

The permanence of information in this EM depends on two things: the continued availability of the paper, and the availability of appropriate access information in STM or LTM. The latter is particularly important, since without some way to find where on the paper information is written, information may be lost as effectively as if it had been erased.

The use of these EM's is a powerful tool in problem-solving; indeed, most of these problems could not be solved without them. Their advantage is that they permit the storage of information, such as code that has been written, that is needed only temporarily or infrequently without either the loss problems of STM or the interference and access problems of LTM. Additionally, they may offer faster and easier information entry and retrieval than LTM does.

Problem-solving in the theory is controlled by a production system. A production system consists of a set of pairs of conditions and actions to be performed when the conditions are met. An appropriate resolution principle is employed to insure that only one set of actions is taken at a time. Executing the actions results in some change in the state of the world so that as the system operates different conditions are met and different actions are invoked. None of the actions involve explicit branching; rather, all control is accomplished through differences in the meeting of conditions and the execution of associated actions.

The production system in the theory is a part of LTM; the conditions it is sensitive to are the presence or absence of certain information in STM. The actions taken when conditions are met change the contents of STM. The theory asserts that the production system is the only internal control mechanism for determining the course of problem-solving; an extensive defense of the suitablity of this particular control structure for modeling human behavior is given in Newell and Simon (1972; p.804). Part of this defense is quoted here:

> 1. A production system is capable of expressing arbitrary calculations. Thus it allows the human Information Processing System (IPS) the information processing capabilities we know he has.
>
> 2. A production system encodes homogeneously the information that instructs the IPS how to behave. In constrast, the standard control-flow system divides program information into the content of the boxes, on the one hand, and the structure of the flow diagram on the other. In a production system this division does not exist, except to the extent that the ordering of productions carries additional information. Production systems are the most homogenous form of programming organization known.
>
> 3. In a production system, each production is independent of the others - a fragment of potential behavior. Thus the law of composition of production systems is very simple: manufacture a new production and add it to the set. This arrangement provides simple ways for a production system to grow naturally from incremental experience.
>
> 4. The production itself has a strong stimulus-response flavor. It is overly simple to identify the two constructs, since productions also have additional properties of matching, operand identification, and subroutine calling that are not apparent in any of the usual formulations of S-R theory. . . . . . Nevertheless, productions might well express the kernel of truth that exists in the S-R position.

5. The productions themselves seem to represent meaningful components of the total problem solving process and not just ood program fragments. This is true in part because we, the scientists sought to define them that way. Nonetheless it remains true that such an organization of meaningful pieces describes the data. . .

6. The dynamic working memory for a production system is the STM (i.e., the memory on which its productions are contingent, and which they modify.) This conception fits well the functional definition of the STM as the collection of information of which the subject is aware at any moment of time. This is not the case with most other program organization schemes . . .in which the relation to directly defined psychological constructs, such as STM is not clear. All these other organizations contain implicitly an unknown amount of machinery that still requires psychological explanation.

   For a production system it remains to specify the matching, the operand definition, the subroutining, and the sequential flow of control on the action side. All these seem amenable to explanation. For instance, each production may possess only a single action operator. In such a scheme the hypothesized action sequences . . . would simply be our short-hand for an iteration through STM in which the output of the first production includes a unique symbol (a linking symbol) to identify the next stage of the action sequence. In this view, the subroutine pointer stack consists of the linking symbols in STM. In such a system almost all the program control apparatus is assimilated to the structure of STM.

   In all events, the gap between program organization and the experimental psychology of immediate memory and processing seems smaller for production systems than for other program organizations.

7. There is an intriguing possibility that a production system offers a viable model of LTM. Possibly there is no LTM for facts distinct from the production system - that is, no basic distinction between data and program; rather the LTM is just a very large production system. If this were the case, the act of taking a new item Into LTM would be equivalent to creating a new production (or productions). . . . . . .

8. A production system, unlike some other programming organizations, offers a nice balance between simulus-bound activity and stimulus-independent activity. The production system itself is totally stimulus bound if by stimulus one means the contents of the dynamic working memory (i.e., STM). All connection between two adjacent actions is mediated by the stimulus so defined. But this stimulus Is per se neither internal nor external, if we take the view that STM is a combination of the internal short-term store and the foveal parts of the visual field (plus of course the symbols that have just been stored in STM upon recognition of other externaal stimuli). If the vast majority of the productions executed are reactions to internally produced symbols, then the system will appear not to be stimulus bound. On the other hand, if almost all productions take as part of their condition an external symbol, then the system will appear to be very stimulus bound. Thus, the overly focused nondistractable character

> of programming models . . . . . is not a structural feature of a production organization, but depends on the particular productions that the system contains.

3.3 Implementation In the Model

In the previous section, the basic structures of the problem solving system, such as the production system and STM, have been discussed in general terms; the following sections discuss the specific way they have been implemented in the present program.

3.3.1 Short-term Memory: Model Implementation

In the Implementation that has been used for this model, STM consists of a fixed number of ordered slots; when new elements are introduced they are placed into the first slot, and each of the other elements is moved down one slot; the element that was previously in the last slot is lost off the end.

During the course of development of this model the number of slots was set at 14. Since this is greater than the classic figure of "seven plus or minus two" (Miller, 1956), a word of explanation is in order. First, what is normally meant by short-term memory span is the capacity for just the stimuli of the experiment; control or intermediate information such as goal or sub-goal markers, temporary variable values, or even the experimental instructions, are not usually included. In this model, however, STM contains plan elements, goals for other actions, and pointers to certain LTM structures which are accessed throughout the course of problem solving. The greater number of slots in STM is necessary to provide space for this information as well as for material more conventionally included in measurements of LTM size.

The number, 14, was selected because, during model development, it was the size at which the system operated effectively. Selection of a size on the basis of evidence from within the protocols, such as forgetting which could be traced to STM overload, would have been desirable. No such evidence was, however, available. This is to be expected in a task such as this since, in the absence of experimental constraints, subjects adopt strategies which minimize the possibility of memory overload (Newell and Simon, 1972).

Besides the introduction of new elements, two other processes alter the contents of STM, element modification and rehearsal. Element modification consists of changing part of an element without altering its position. In rehearsal, an element already in STM is moved into the first slot from some other position; the remaining elements are each moved down one slot until the empty space is filled.

3.3.2 Production System: Model Implementation

The production system used in the model is one of the variants possible in the PSG system (Newell, 1973). In this variant, the invoking conditions are always tested in a fixed order, and the first production which is true is executed. Thus, no contention problems arise if the conditions of more than one production are met.

The invoking condition for a production consists of a specification of one or more items which must be present in STM for the condition to be true. If a condition does contain more than one item, then the items are treated conjunctivly, and all of them must be present for the condition to be true. It is also possible to indicate (by preceeding them with the special symbol, *ABSENT*) that items must be absent from STM for the condition to be true.

Specification of an item as part of a condition can be done by giving the exact item; alternatively, it is possible to use variables as part of the specification so that an item can be described in general terms. Variables can be of two types; those which match parts of items on the basis of their structure and those which match on the basis of content. The first type of variable is indicated by the presence of the symbols, *ATOM*, *LIST*, *ANY*, and *REST*, in the description of the item; for example, *LIST* matches any list which occurs at the corresponding point in the item. The second type uses the symbol, *CLASS*, followed by a list of information that may appear at that point in the item specification; if it were desired to match either an A,B or C in the item, then these three symbols would appear following *CLASS*. Using variables of the first type, a plan element such as "a PLAN-ELEMENT which calls for FINDing the FIRST POSITIVE in the LIST OF NUMBERS" may be specified as "any element which begins with a PLAN-ELEMENT." Variables of the second type permit specifications such as "any RESULT, PLAN-ELEMENT, or GOAL which contains a FIND."

Matching within each individual condition proceeds on a first to last basis, and each element in STM matches at most one element in the condition. This means that if the condition is "an element which is a CURRENT-GOAL" and there are two elements in STM which are both CURRENT-GOALs, then only the first of them will take part in the match.

The combination of first-to-last matching with the ability to state match conditions in general terms has an important psychological consequence. Consider a situation in which the subject is exposed to (CURRENT-GOAL FIND) followed by (CURRENT-GOAL LOOP). Because STM has a first-in first-out structure, the two items would appear in STM as:

(CURRENT-GOAL LOOP),(CURRENT-GOAL FIND)......

If the production to remember what the subject was shown last had as its invoking condition, "anything which begins with a CURRENT-GOAL", then the subject would recall (CURRENT-GOAL LOOP), not (CURRENT-GOAL FIND). Behaviorally, this can be interpreted as interference between the two items.

With the exception of the COMPARE-EFFECT function, discussed in the section on Processing Mechanisms, the action part of each production is an unconditional sequence of operations. No branching takes place among them.

Figure 3.3.2.1 shows the action operators that are used in the productions and the knowledge structures which they affect. (Most of these knowledge structures are discussed in the next section.)

Action Operators and the Structures They Affect

Actions	Structures
REHEARSE	STM
REPLACE	STM
NEW-ELEMENT	STM
NEWMEANING	MEANINGS
GETMEANING	MEANINGS
CODE-GEN	STM
ADDCODE	CODE
MODCODE	CODE
COMPARE-EFFECT	STM

Figure 3.3.2.1

An example of an actual production used in the system is:

```
NEW-CODE-1.

CONDITIONS:
1.  (NEW-CODE *ANY*)
2.  (CODE)
3.  (OLDCODE)
4.  (PLAN-ELEMENT *REST*)
5.  (MEANINGS)

DESCRIPTION OF ACTIONS:
1.  REPLACE (NEW-CODE *ANY*) BY (ADD-ON *ANY*)
2.  REHEARSE  (CODE)
3.  REHEARSE  (PLAN-ELEMENT *REST*)
4.  REHEARSE  (MEANINGS)
```

The CONDITIONS are patterns for items which must be present in STM for the actions to be taken. A complete listing of all the productions is given in Appendix 5.

3.3.3 Other Long-term Memory Structures

The power of production systems as a programming device and their psychological relevance suggest the attractive possibility of constructing the entire model as a single production system. This approach has not, however, been followed here for this reason: For the model to accurately reflect behavior, the contents of LTM must be modifiable both within a single problem and across problems to reflect the changes in knowledge caused by such things as the creation of new code-variables, (i) assigning effects to new code, and the construction of new plans. While a production system which could modify itself would have the necessary properties, the difficulties involved in building productions which create other productions are extreme. Among the problems to be solved are finding strategies for deciding when a new production is to be created, for specifying what the conditions and actions of the new production are to be, and for inserting the new production at the right place in the list of productions. There is also the programming problem of building a system which can modify itself.

Because of this problem, types of information which must be modified within a a problem or across the problems in the set are represented in two structures outside the main production system, MEANINGS and CODE. They are accessed via access

----

(i) Both the model itself and the programs written by the subject contain such things as variables, expressions, etc. To avoid confusion, when parts of a program written by the subject are being referred to, the terms will be preceeded by "code" as in code-variable, code-expression or the general term, code-quantity. Use of these terms not preceeded by "code" will refer to the model program.

functions called from within the production system; additionally, some of the processing on these structures is done by special processing functions that are also called from the production system. Both of these structures and their access and processing functions are discussed in the following sections.

3.3.4 MEANINGS

As writing of the subject's program proceeds, a body of information about the program gets built up. Some of this information is contained in the code itself, but much of it, such as the meanings of code-variables and code-labels and the effects of pieces of code, cannot be retrieved from the written code alone and is used over much too long a time period for it to remain in STM, at least in an un-encoded form. The structure outside the production system that contains this information is called MEANINGS.

MEANINGS is organized as a collection of attributes and their values, one set for each variable or expression in the subject's code. Examples of the attributes include the TYPE of the expression - with values of pointer, label, array, etc., and the NAME that is actually used for it in the FORTRAN program, e.g., FRSTOD, L, I, or NEXT. The values that these attributes may have can be either absolute or they can be defined relative to another attribute or quantity. An example of an absolute value would be FRSTOD for the NAME attribute of a variable used as a pointer to the first odd number. An example of the use of a relational value might be in defining the value of one variable as being a pointer to another variable, as in a variable which means "pointer to the positions in array L."

Not all attributes are necessarily present for each code-quantity or even for all code-quantities of the same type; while a LENGTH attribute would be used with a code-quantity of TYPE "array," (i) it would not be used for a code-quantity of TYPE "pointer" or even recursively with another quantity of TYPE array. What determines assignment of particular attributes to a code-quantity is whether use is made in the protocol of a value of the attribute for the quantity; for example, a code-quantity of TYPE ARRAY would have an attribute of LENGTH only if the length of the arrsy were actually used in the protocol.

Instead of assigning an attribute only if it was used in the protocol, an alternative strategy would have been to define generic classes of quantities which each would have a set of required and forbidden attributes; for example, a code-quantity belonging to the class, <pointer-variable>, might always have the attributes, NAME and VALUE, and never have the attribute, LENGTH Whenever a code-quantity belonging to this class was used in the model, all of these attributes would be given values and placed into MEANINGS. Such a strategy lends itself to the sort of procedural embedding of knowledge used in systems such as PLANNER (Hewitt, 1972); additionally, it probably has considerable psychological validity, since it is a likely supposition that programmers do, indeed, associate whole families of attributes with the type of a code-quantity. The reason for not using it in this model is primarily one of research strategy: using generic classes means that the attribute sets must be defined before the rest of the model is built and that the whole model must be revised each time these sets are changed. The strategy which has been followed, of adding attributes as they appear in the protocol, is

----

(i) Readers familiar with the issue of "type" in progrmming languages should be aware that the attribute, TYPE, refers to the way in which the subject conceptualizes the use of a code-quantity, not to its formal type. Under this usage, the attribute, TYPE, might still be needed for modeling programming in a "typeless" language.

adequate for developing the rest of the model and does not require this constant revision.

An example of an attribute that is frequently present for variables and labels is called MEANS; its value is the type of information that programmer intends the variable to contain or the location in the programming a label is represent. Thus, in a program to find the first odd number in a list of 100 random numbers, there might be a variable of TYPE "array," LENGTH "100," and NAME "L" with the value of MEANS being "random numbers."

The attributes that are actually used and a description of their values are given in the following table:

Table 3.3.4.1
Attributes Used in MEANINGS

Attribute	Description of Values
NAME	FORTRAN name used in program
TYPE	Type of a quantity such as POINTER, LABEL, or ARRAY
MEANS	Intended meaning of a quantity, such as "pointer to odd numbers"
LENGTH	Length of an array
BEGINNING	Beginning of particular information in an array, usually 0, since the information usually starts at the beginning of the array
EXPRESSION	Instantiated template
EFFECT	An effect assigned to an instantiated template

At the lowest level, all access to MEANINGS are performed by two functions, GETMEANING and NEWMEANING. These are used by the production system and by the CODE-GEN and CODE-EL routines (described later) to retrieve and add information to MEANINGS. The first of them, GETMEANING, takes two arguments, the name of an attribute whose value is desired and a list of other attributes and their values which belong to the same set. Which particular Information is included in this list is determined by the call to the function, not by GETMEANING itself. GETMEANING then searches MEANINGS for a set containing both the desired attribute and the known attributes and values. If one is found, the value of the attribute is returned.

As an example of how GETMEANING works, consider a situation in which an array called RAN, which contains random numbers, has already been used in the program. Now it is to be used again, and the particular name, "RAN", which is used in the program, must be retrieved again. In the protocols, this might be indicated by something like, "Let's see, I need to use that random number array again. What did I call it?" Depending on the situation, in modeling this behavior, either CODE-GEN or CODE-EL would issue a call to GETMEANING. The first argument would be NAME, since it is the name of the array that is needed; the second argument would be the list: MEANS-"random numbers", TYPE-"array." The function would return "RAN" as its value.

Additions or modifications to MEANINGS are made by the NEWMEANING function which is called as a direct action of the production system. The first argument is a list of new attributes and their values which are to be added to the set for a particular code-quantity. The second argument, if present, is a list of attributes and values that already belong to the code-quantity set. The particular information present in this second argument depends on which production calls it. The second argument is used in the same manner as the second argument to GETMEANING, to locate the set belonging to a particular code quantity. Once the set is located, the attributes and values in the first argument are added to it. Thus, in the previous example, if it were now decided that the array in question were 100 elements long and this information were to be added to MEANINGS, the call to NEWMEANING made by the production system would have as its first argument, LENGTH-"100", and as its second argument: NAME-"RAN", TYPE-"array", MEANS-"random numbers."

To add an entry to meanings for a completely new code quantity, NEWMEANINGS is called with no second argument. The attributes and values in the first argument are then entered into MEANINGS as a new set.

In evaluating the psychological implications of this model, the first thing to be noted is that, in retrieval of information about a particular code quantity, the quantity is specified only by the list of its known attributes and values. This, in turn, Implies that information about the same quantity may be retrieved by a variety of different routes. Given a quantity of TYPE-"array", NAME-"RAN", and LENGTH-"100", which MEANS-"random numbers", the NAME may be retrieved either by specifying it as being of TYPE-"array" and LENGTH-"100" or as MEANS-"random numbers" and TYPE-"array". This ability to retrieve knowledge via a variety of routes using whatever information is available is a useful one for modeling a set of protocols since, in them, no single type of knowledge is consistantly used for retrieving information about code quantities.

The way in which code quantities are specified for retrieval also allows representation of one common type of interference phenomenon. Suppose that several different quantities have some attributes in common. If only a few of the attributes of a desired code-quantity are specified and if these attributes are common to several code-quantities, then the wrong code quantity may be retrieved. This provides a mechanism for modeling those errors in which, out of several similar variable names or expressions, the wrong one is selected.

3.3.5 Code: Basic Organization

CODE, the third major LTM knowledge structure in addition to the production system and MEANINGS, is information about how to access an EM, the paper containing the code that the programmer has already written. For the following reason, It Is quite likely that very little of the actual code remains accessible in LTM once it has been written out on paper: when the subject in this study wanted to re-write or re-use pieces of already-written code, longer than a line or so, he almost always had to find and read the written code, indicating that he was unable to recall them directly from memory. Any use, modification or correction to code which has been written must therefore retrieve the code from the paper EM; and the LTM must contain the information necessary to perform the retrieval.

Using a representation of eye-movements, an explicit model for retrieval from an EM has been built for a seriation task (Newell, 1972). Since eye movement or other perceptual data were unavailable, no attempt has been made to be as explicit in this case. Instead, a simplified structure for access to the written-code EM has been assumed: it is always searched linearly and exhaustively, most recent code first. While this is inaccurate for those situations in which some sort of index into the EM is used, it is reasonably close to the actual situation in many cases, and, in those situations in which it is not correct, its effects on the model can be compensated for in other ways.

While CODE has been defined as a memory that gives access to the written code, internally, it consists simply of a listing of the lines of code that the programmer has already written. To be consistant with the definition given previously, each of these lines should be interpreted as a pointer to the "real" line that the programmer has written on paper. It can altered by two functions, ADDCODE and MODCODE, which, respectively, add new code and modify existing code by replacing one piece of code with another. Deletion of code is treated as a type of replacement. Retrieval of information from code is done by a function, RETRIEVE-CODE, which uses information from MEANINGS about the effects of code to accomplish retrieval; it is discussed more fully in the section on Adding and Modifying Code.

3.4 Knowledge Representations in the Model

The preceeding section describes the basic knowledge structures of the model. The following section describes the way knowledge is represented within these structures.

3.4.1 The Plan

According to this theory of programming, a plan consists of a sequence of functions which must be performed in order to achieve the desired effect of the program. In the model, a plan is considered to be part of the production system; pieces of it are placed one at a time by a production into STM for coding.

In the protocols, a single, functional language is used to talk about both plans and the effects of pieces of code. This is reflected with in the model by using a single notational system to represent both. For ease of understanding, only the important characteristics of this notation is presented here; full details are available in Appendix 4. The general form that plan elements expressed in this notation take is:

<function to be performed> <operands>

A few examples of actual plan elements, with explanations, are given below:

```
a. (IF ((EVEN PARITY) ((LIST OF NUMBERS)(POINTER (NEXT ODD)))
       (GOTO (LOOP END))))
```

"If the element in list of numbers which is pointed to by the pointer for the next odd number is even, go to the end of the loop"

```
b. (ORDER (LIST OF NUMBERS))
```

"Order the list of numbers."

```
c. (FIND-EXISTENCE ((FIRST POSITIVE) (LIST OF NUMBERS)))
   (BEGIN! (OTHERWISE))
   (SET (CORRESPONDING-ELEMENT (AUXILLARY ARRAY)
                       (VARIABLE (LOOP-INDEX)))
     (VARIABLE (LOOP-INDEX)))
   (END!)
```

3

```
(END! (FIND-EXISTENCE-LOOP-THROUGH))
```

"Loop through the list of numbers until the first positive is found. If a numbers is not positive, then set the corresponding element of the auxillary array to the value of the loop index."

```
d. (FIND-AND-SWAP ((MULTIPLE) (LIST OF NUMBERS)
                              (VARIABLE (LOOP-INDEX)))
     (ARRAY-ELEMENT (LIST OF NUMBERS)
                         (VARIABLE (INNER-LOOP-INDEX))))
```

"Loop through the list of numbers looking for multiple of the array-element pointer to by the inner loop index"

The elements beginning with BEGIN! and END! are special marker elements. In some situations it is necessary to indicate that a group of these items are to be performed together; examples might be to show that all the items in the group belong inside the same loop or that they are part of the same branch of a conditional. For this purpose, these special marker elements, named after the BEGIN and END in ALGOL, are provided which may be placed before and after sets of item to indicate that they belong together in a group. (i)

A final comment about this notation as applied to plans is that it makes no distinction between plan elements which lead to the generation of actual program code, for example, "set the pointer equal to the index of the first odd number found," and those which only result in the establishment of data representations, such as, "create a pointer to keep track of the location of the first odd number." This use of a common functional notation to represent both types of plan elements is true to the way both types of plan elements behave in the protocol.

(i) This simple structure is adequate for all the protocols used in this study, probably because the problems are simple ones; more complex problems may require the use of sub-lists or trees to represent plans.

3.4.2 Templates and Effecis: General Structure

Since the plan itself is presumed to be language-independent, the information about the syntax and semantics of the language in which t e code is actually written must be contained in the production system. For syntactic information, this is done by means of structures called coding templates which are formally equivalent to a Backus-Normal form definition of the language, using very high-level primitives and very few recursion slots. Each code template consists of a small segment of code - at most 3 or 4 lines - specified as a mixture of three types of information: (1) actual code elements; (2) descriptions or specifications of code elements that are to be inserted in the code; (3)and parameter slots which will be replaced by descriptions or specifications of code elements at the point when the template is actually used.

The actual code elements that appear in the template are primarily keywords and separators in the particular programming language; for FORTRAN, examples might be DO, GOTO, commas and parentheses. Descriptions or specifications are used for the names of labels and variables, and for nested expressions; examples might be "label for the loop scope" or "pointer to the first odd."

Parameter slots allow the templates to have wider generality by delaying the description or specification of quantities until the template is actually used. An example might be a template for setting two simple variables equal to each other; descriptions of the specific variables would only be plugged in at the time the template was used.

When a code template is invoked in response to the content of a plan element, the elements of it are processed one by one by a code generation function. Actual code elements are handled by having them leave appropriate traces in STM and in CODE, indicating that they have been added to the written program. The handling of slot descriptions depends on whether the code-quantity or expression has occurred previously in the program. If it has, then MEANINGS already contains the necessary information about it, including its FORTRAN name; the name is then retrieved and treated as an actual code element. If it is not, then a goal is created for creation of the desired code-quantity. After the goal has been satisfied and information about the code-quantity or expression has been placed into MEANINGS, then its name is also used as actual code.

In some cases what is described in the slot will be not just a simple variable name or expression but a more extensive piece of code. When this occurs, another code template may be invoked inside the first one, a process which can be nested arbitrarily deeply.

A typical template for a DO loop might look as follows before the parameters are plugged in:

```
DO "label for (parameter #1) loop" "variable for loop index" =
  "begins at (parameter #2) loop" , "ends at (parameter #3)"
```

If the loop were to be used to go through an array, the template might look as follows with the parameters substituted:

```
DO "label for go-through-array-L loop" "variable for loop index"
  = "begins at beginning of L array" , "ends at end of L array"
```

Elements shown in bold-face, such as DO and =, are actual code elements which will be put directly into the final code. Quoted items are descriptions of variables or expressions; to generate the final code, the mechanisms that have been described previously must be used to substitute the actual elements for the description. When this is done, the code that is generated might look like:

```
DO 100 I=1,100
```

As specified in the theory, an effect is associated with each piece of code as it is written. This linkage is accomplished within the production system itself. As each template is used, one of the actions of the production is to place a copy of it with free parameters instantiated into MEANINGS under the property, EXPRESSION; the associated effect is put under the property, EFFECT. When code generation from the template is completed, the effect is retrieved and placed into STM. Since it is still a part of the MEANINGS structure, the effect also remains available for retrieval on other occasions, such as while checking code or doing initializations.

Since in the protocols, subjects use the same sort of functional language for both plans and effects, as has been mentioned, the same notational system that was described in the section on plans is used in the model for both. The primary distinction that may be made between the two is one of content; effects are usually much more specific as to where and how the function is actually accomplished. A few examples of effects, with explanations, are given below. It is worthwhile to compare them with the plan examples given previously.

a. (BRANCH-IF ((EVEN PARITY)((LIST OF NUMBERS)(POINTER (NEXT ODD)))
(GOTO (LOOP END))))

"If the element in the list of numbers which is pointer to by the pointer for the next odd number was even, this branches to the end of the loop."

b. (ORDERED (LIST OF NUMBERS) (LABEL (ORDER-LIST-LOOP-END)))

"At the label for the end of the list ordering loop, the list is ordered"

c. (FOUND ((FIRST POSITIVE) (LIST OF NUMBERS))

3

```
                    (LABEL (POSITIVE FOUND))
          (ARRAY-ELEMENT (LIST OF NUMBERS)
                         (POINTER (FIRST POSITIVE))))
```

"At the label, the first positive has been found and is pointed to by the pointer."

3.5 Processing Mechanisms

The model that has been presented so far may be characterized as follows: The individual elements of the plan, expressed in the functional plans-effects language, are placed one by one (by a production) into STM. Productions then fire off which attempt to convert these plan elements into to code using the information about language syntax contained in the templates. As each piece of code is created the production system assigns to it an effect which may then serve as part of the stimulus for further productions. Information about the actual written code is accumulated in an LTM structure, CODE. Information about the meanings which have been assigned to variables and labels and the effects which have been assigned to code are accumulated in another LTM structure, MEANINGS.

In the course of carrying out this cycle the production system uses five special mechanisms which supply it information or operations that are not otherwise represented in the model. The following section describes each of these mechanisms and explains why it is used.

3.5.1 Code Generation and CODE-GEN

The actual function of converting a template into code is performed by a fucntion, CODE-GEN, which is called as an action by various productions. The use of a separate function for this operation, rather than performing it within the production system, is more a consequence of the program structure than of psychological assertions of the model. The encoding of a template requires a great deal of use of the information contained in CODE and MEANINGS. Direct retrieval from these structures by the production system is involved and difficult to program properly; placing these actions in a separate function greatly reduced the effort involved in system construction.

CODE-GEN relies heavily on two other functions, PUTARG and CODE-EL. The first of these, PUTARG, is responsible for instantiating the parameters in the templates. It does this on a purely positional basis, assigning the first value to the first parameter, the second to the second, etc. The only check it makes on the values is that they correspond in number to the parameters.

CODE-GEN takes as its primary argument an instantiated template and tries to convert this template into code and return its associated effect. It begins operation by taking the elements of the template one at a time and passing them as arguments to another function, CODE-EL. If the template element is an actual piece of code, such as a DO or comma then CODE-EL returns it unaltered. If the template element is a variable or expression description, then CODE-EL attempts to use the GETMEANING function to retrieve from MEANINGS the actual code corresponding to the description. If it succeeds, it returns the code as its value. If the variable or expression is an entirely new one, then actual code for it will not be present in MEANINGS. When this happens, CODE-EL fails and returns NIL.

CODE-GEN accumulates the actual pieces of code until the template is exhausted or until CODE-EL fails. When the former occurs, CODE-GEN places both the accumulated code and the retrieved effect of the template into STM as NEWCODE and EFFECT elements respectively.

If CODE-EL fails, then CODE-GEN takes three actions; it places the code that it has accumulated so far into STM as a NEWCODE element; it creates a GOAL element in STM for the template item that caused the failure; and it creates a CODE-GENERATION STM element. The latter contains a copy of the template that CODE-GEN was encoding and a pointer to the position in it at which the CODE-EL failure occurred; it's function is to permit a return to generating code from the template once the goal has been satisfied and is the equivalent of a higher node in a goal tree.

A fairly frequent occurence in the protocols is that the programmer writes some new code that is a modification or replacement for some code that has been written previously. In most cases, information about the old code is still present in STM and can be used by the production system to make the modification or replacement. In a few cases, though, the old code has been written long enough ago so that the information about it is available only in MEANINGS and CODE. CODE-GEN checks for this situation by using a function, SIMILAR-EFFECT, which searches MEANINGS for similar, but not identical, effects to the effect of the current template. "Similarity" is defined as:

1. Having the same verb as the current effect.
2. Having the same first argument as the current effect.

When an old effect meeting these criteria is found in MEANINGS, CODE-GEN responds by using a function, RETRIEVE-CODE, to retrieve the code that was generated from the older template. A special element, (OLDCODE), which is a pointer to this existing code, is then placed into STM along with each NEWCODE element. This OLDCODE element is used by the production system to produce appropriate modifications of the older code.

A flow-chart of the operation of CODE-GEN is given in Figure 3.5.1.1.

Flow Diagram for the CODE-GEN Function

```
1.  Get effect of template.
    FAIL, there is no effect.
     1.1 Report error and QUIT.
2.  Get next code element.
    FAIL, there are no more code elements.
     2.1  Test for accumulated code.
          YES, there is some accumulated code.
           2.1.1  Perform write-out-new-code.
          NO, there is no accumulated code.
           2.1.2  Perform end-of-template.
3.  Repeat until failure occurs:
    3.1  Call CODE-EL with the next code element as argument.
    3.2  Add what CODE-EL returns to the accumulated code.
4.  After CODE-EL fails, test for accumulated code.
    YES, there is accumulated code.
     4.1  Perform write-out-new-code.
     4.2  Create a CODE-GENERATION element.
     4.3  QUIT
    NO, there is no accumulated code.
     4.4  Create a GOAL element.
     4.5  Create a CODE-GENERATION element.
     4.6  QUIT
```

```
Write-out-new-code.
1.  Test if a similar effect already exists.
    YES, it does.
     1.1 Retrieve the code that went with it.
     1.2  Set OLDCODE as a pointer to this old code.
     1.3  Place an (OLDCODE) element into STM.
2.  Place a NEW-CODE element containing the accumulated code
    into STM.
3.  QUIT
```

```
End-of-template.
1.  Place any code conditions associated with the template
    into STM.
2.  Place the EFFECT of the template into STM.
3.  QUIT
```

Figure 3.5.1.1

Psychologically, the way in which this system generates code has important implications. First, a goal for the creation of a variable or expression is generated only if the information is not already available from MEANINGS. The system is, thus, a self-modifying or learning one which alters its own behavior over time.

Second, in coding the elements of a template, the system proceeds in a strict, first-to-last fashion. Whenever a subgoal is created, whether from the template itself or from some prior subgoal, it is attempted immediately. If any subgoal in a chain fails, coding of the entire template fails Thus, there is no provision for defering goals, and no back-up in a subgoal chain occurs.

3.5.2 Comparing Plans and Effects

comparison between a part of a plan and the effect assigned to code generated from the plan is an important feature of this model. These comparisons are performed by the COMPARE-EFFECT function. A plan element and an effect are considered to correspond if:

1. The function named in the effect indicates completion of the function required by the plan. For example, LOOPED-THROUGH as an function in an effect would indicate completion of LOOP-THROUGH in a plan. This correspondence is determined from a table of equivalents in COMPARE-EFFECT.

2. The operands of the function in the plan must also be present in the effect. Continueing the example, if LOOP-THROUGH has LIST-OF-NUMBERS as an operand, then LOOPED-THROUGH must also have the operand for the two to correspond.

If a plan and effect meet these criteria, they correspond. Any additional details present in the effect are ignored.

COMPARE-EFFECT is called as part of the action side of productions. Depending on what it returns, either a MISMATCH element is placed into STM or both plan and effect elements are marked as OLD. A non-branching production equivalent to the use of this function would have been to do the comparison within the production system. The drawback of this approach is that it would have required a large number of very specific productions; to avoid this, the branching function departure from a pure production structure was used.

3.5.3 Rehearsal and the LOOK-AT-CODE Mechanism

Because of the first in - first out operation of STM, the model asserts that STM items which are not rehearsed are eventually lost off the end. To prevent loss of needed information, either STM must be large enough so that an item is retained until needed or a rehearsal strategy must be adopted which keeps the item at the front of STM. The conditions under which rehearsal take place are, therefore, important in determining the behavior of the model for a given STM size. The basic prerequisite imposed by the model is that rehearsal of an item may take place only when the presence of the item is part of the invoking condition of a production; the rehearsal is then accomplished as one of the actions of the production. This approach has the desirable property that knowledge in STM which is actually used or "attended to" is rehearsed and remains available over time while unused items are eventually lost - "decay" - as a function of interaction.

While this mechanism works very well for most situations, it does run into difficulty because of the structure of the CODE EM. If CODE were part of the production system, then certain information would be rehearsed as a function of using it as part of the invoking conditions of productions; for example, the information that a DO loop was still open might be rehearsed by a production which caused the programmer to look at a DO statement that he had written previously. Because CODE is not part of the production system, either a very large STM must be used or a special rehearsal mechanism must be provided. Since a very large STM might distort other parts of the model, the latter course has been followed.

The mechanism that has been selected is to provide a special STM item, (LOOK-AT-CODE), whose presence is used to indicate certain accesses of the CODE EM. When it occurs in STM the element may be responded to in two ways. One of these is to include it as part of the invoking conditions for productions which rehearse appropriate STM items, usually those associated with code that has already been generated; for example, one such production would result in the rehearsal of any item associated with an open DO-loop condition.

The other response to the LOOK-AT-CODE item is similar to Newell's (1973) CALL operator. A production sensitive to the presence of the item calls a special action element which results in an interruption of the program and the return of control to the user's console. The user may then rehearse items or provide whatever other actions are necessary and then continue program execution. This mechanism is provided for those situations in which behavior is dependent on the contents of the CODE EM and which can't otherwise be included in the production system. An example might be behavior which is caused by the programmer's noticing a certain variable name and discovering that he has failed to initialized it. Since this mechanism is necessary only for those few special cases, its use in the program is relatively infrequent.

As with all other STM elements, LOOK-AT-CODE is placed into STM by the production system. The production which does this must be sensitive to situations in which it is likely that CODE EM has been accessed. The situation that has been chosen is the completion of coding of a template. An alternative choice would have been to invoke the production every time a new piece of code was generated; however, since creation of new bits of code is a very frequent event, this would have meant that the program would have spent most of its time processing LOOK-AT-CODE items and that considerable space in STM would have been taken up by them (with the effect of pushing other items off of the end). The effect of this compromise is a slight loss in the sensitivity of the system to some of the finer aspects of behavior driven by the contents of CODE. An example might be a case in the protocol in which an error is made and responded to before the coding of an entire template is completed; because of this compromise, the model would only be able to make the correction after completion of the entire template.

3.5.4 Modifying Code and HOW-MODIFY

The usual action of the production system as each new piece of code is generated is to add the new code onto the end of the CODE EM using the ADDCODE function. As was discussed briefly in the description of the CODE-GEN function, in may cases the new code is intended to modify some previous code and is to replace the previous code or be inserted before it. Indication that this is the case may come from the production system (frequent case) because it has just discovered that a piece of code gives a wrong effect or via the SIMILAR-EFFECT function in CODE-GEN (infrequent event). In both cases the following process takes place, either within CODE-EL or as an action of the production system: The effect of the older code is passed as the argument to a function RETRIEVE-CODE, which uses GETMEANING to retrieve the instantiated template which was used to create the code in the first place. Once the template is retrieved, RETRIEVE-CODE uses CODE-EL to turn the template into code again; since the code has been created previously all the necessary information to do this is already in MEANINGS. The OLDCODE element is then set as a pointer to this old code and placed into STM. (The actual value of the identifier, OLDCODE, is all the code in CODE up to the piece that is to be replaced or modified.)

To actually modify the code, a special function, HOW-MODIFY, is provided. Motivation for creation of this function was that there are many ways in which code can be modified. If there is enough room, the new code may be inserted before the old code; alternatively, the old code may be crossed out and the whole thing recopied. If the old code is written lightly enough, he may just write over it. Because the action taken depends on such things as how much space he has left on the page, how dark his pencil lead was, etc., no attempt was made to simulate this behavior in the protocol. Instead, as the newer code is created, a production sensitive to the presence of both OLDCODE and NEWCODE elements in STM is fired. One of the actions of this production is to call HOW-MODIFY which returns control to the operator's console so that he may specify whether the new code is to be inserted before, replace, or be inserted after the older code. Other productions then use this information to call on ADDCODE and MODCODE to make the appropriate changes in CODE.

3.5.5 Generating Names: NEWQUAN

In order for the model to satisfy GOALS for variables and labels, it must have available the FORTRAN names that are to be used in the program. The productions that satisfy these goals obtain these names by calling the function, NEWQUAN. The first argument to this function is the meaning of the code-quantity whose name is desired; the second argument is the type of the code-quantity. A third argument is present only if the code-quantity is of type LABEL. Since labels in FORTRAN are numbers, it indicates the relative increment of the new label over previous ones.

In the protocols, the names given labels seem to follow a regular pattern. Labels for loops containing only a few calculations increase by 10 for each new label; that is, successive labels of this type would be 10,20,... Labels for loops containing many calculations, particularly other nested loops, increase by 100. When labels of the first type occur inside labels of the second type, they increase by 10 starting from the outer label, as in 110,120,.... Finally, there are a few occurances of very large loops that contain large segments of program inside them; they have numbers like 500,1000 etc.

To generate new labels, NEWQUAN maintains two counters, OLD-SMALL-LABEL and OLD-LARGE-LABEL, which are, respectively, the values of the last small and large labels to be used. When a new small label is requested, the first counter is incremented by 10 and the new value returned as the new label. Similarly, for a new large label, the second counter is increased by 100 to get the next label; at the same time, though, the small label counter is set to the new value of the large label counter so that any successive new small labels will be larger than the new large label. Finally, for new very large labels, 500 is added to the current value of the large-label counter.

Unfortunately, the generation of names for variables by the subject is not nearly as systematic as that for labels. Instead of attempting to generate them, NEWQUAN uses one of two methods to retrieve them. The first is to search NAMES-LIST, a list of meaning-names pairs that is "plugged into" the program whenever these names will be used. If a needed meaning-name pair is not found in this list, then the second method which NEWQUAN uses is to ask at the user's console for the names. The generation of names is, thus, not really part of the model, but is effectively a parameter to it.

3.5.6 Symbolic Execution and the Structure of the Model

In the presentation of the theory in Chapter 1, it was asserted that creation of program code was accomplished by symbolic execution; yet, the model just presented does not contain symbolic execution as an explicit construct - i.e., there is no single routine that can be pointed to as the symbolic execution routine. Instead, symbolic execution is present implicitly, as a function of the interaction of the routines and structures of the model. This can be illustrated using the example of code generation for the problem of printing out all the odd numbers ina an array. The plan for the program might begin with "loop through the array." A production sensitive to this plan would fire off which would retrieve an appropriate template, instantiate it, place information about it into MEANINGS, and finally call CODE-GEN to code it. This process is equivalent to the laying-out of code in symbolic execution. When the template is coded, its effect is placed into STM, corresponding to the assigning of effects or consequences in the theory. Finally, the presence of the effect in STM serves as part of the invokiing conditions for a production for the next step in the plan, corresponding to the statement of the theory that effects or consequences of one piece of code are used to generate the next piece. Viewed across the entire system, this example makes clear that symbolic execution is an inherent property of the model.

4

Application of the Model to the Protocols

Development for Segments of the Protocols

Using the structures and mechanisms described in the previous chapter, the model was developed to fit the coding behavior seen in segments of four protocols. These four segments were selected from the 42 coding segments in the 23 protocols in the following way: First, of the 23 protocols, a set of 7 (WILLIAM, CARL, PAUL, ROBERT, KEVIN, FRANK, ALLAN) were set aside for use in verifying the model; they were selected because, as a set, they had approximately the same distribution of solution times as the entire set of 23.

A second criterion was then applied to the coding segments in the remaining 16 problems: All those segments which consisted primarily of symoblic execution of code that had been written previously, as would be the case, say, in going back to do initializations, were eliminated. The reason for doing this was that behavior in those segments would depend heavily on retrieval from the CODE EM. Because of the simplified structure used for this EM, this would necessitate a great many calls to the operator's console for information. Since the information supplied would have a great deal of influence on system behavior, evaluation of the behavior produced by the model alone would be made more difficult. From the segments remaining, the four segments from four different protocols were chosen; particular selection were based on the individual characteristics described in the discussion of each segment.

A single production system and set of templates (i) was used for all four segments;

(i) i.e., Some productions and templates are used in all four segments, though there are idiosyncratic productions.

this production system and the templates are shown in Section 1 of Appendix 3. Different plans were used for each individual segment. Additionally, at the beginning of each segment, MEANINGS was assumed to contain only the information about the arrays, L and M. The following section discusses the behavior of the system for the first problem in complete detail and an overview of system behavior for the other 3 problems.

4.1 Problem RICHARD

Problem RICHARD was the eleventh program done by the subject. It was one of the shorter problems in the set, taking only 13.3 minutes to write and 3.0 minutes to debug and run. The problem was to find all the odd numbers in the array, place them at the beginning of the array, and set the elements in M corresponding to the final positions of the odd numbers to 1. The complete protocol is given in Appendix 2.

It was selected for modeling primarily on expositional grounds. The problem is easy to understand, is not subject to many conflicting interpretations and is readily solvable by most people with programming knowledge. The subject takes only a moderate amount of time to solve it, and his solution in the protocol is quite easy to follow. Since it is an easy problem to explicate (though not necessarily to model) it was chosen as the first problem to be modeled by the program.

4.1.1 Planning in the Protocol

Planning begins very quickly after the subject receives the problem description; it takes place in lines 8 through 12 of the protocol (i.e., about 30 seconds after he receives the problem description):

S8:GO THROUGH THE ARRAY
S9:DETERMINE IF A NUMBER'S ODD OR NOT
S10:HAVE A POINTER TO THE LAST PLACE WHERE THERE'S NOT AN
S11:TO- IF IT'S RIGHT AT THE BEGINNING THEN YOU KNOW HOW FAR YOU HAVE

S12:ON ODD AND HOW FAR YOU HAVE ON EVEN

Since this is the only identifiable planning behavior seen in the protocol, it is assumed that planning was completed in this segment and that this same plan was used without modification throughout the writing of the entire program. From this segment it can be determined that the plan consists in part of looping through the array, testing each number, and keeping a pointer to the last position at which a non-odd occurs. In lines 16-18 he says:

S16:IT'S A POINTER FOR NEXT ODD
S17:NEXT ODD
S18:AND THE OTHER ONE IS JUST GOING TO GO THROUGH THE ARRAY SO

This indicates that the plan actually consists of keeping two pointers, one for the position at which the next odd is to be placed and one which goes through the array pointing at the next element to be tested. Finally, as he is checking over the program in lines 52-59, he comments:

S52:SO THAT ALL ODD NUMBERS ARE AT THE BEGINNING
S53:PLACE ONES CORRESPONDING POSITIONS IN M
S54:AND THAT
S55:INITIALLY IF I EQUALS ONE
S56:THEN WE DON'T EVEN HAVE TO WORRY ABOUT IT
S57:AHH, SO WE'LL JUST DO THAT SWITCH
S58:RATHER THAN TEST IT EVERY TIME THROUGH THE LOOP
S59:AND THAT JUST WON'T HURT ANYTHING

From these comments and from the code he actually writes, it may be inferred that, once he has found an odd number, he intends to increment the pointer to the next odd, swap the odd number with the element pointed to by the pointer to the next odd, and then set the corresponding element to 1.

Re-written as a sequence of actions to be performed, an informal statement fo the plan would be:

```
1. Create a pointer to the next odd number, starting out at the beginning of the
     array.
2. Loop through the array.
3. Test each element to see if its odd.  If It Is,
      Increment the pointer to the next odd.
      Swap the element it points to with the element that was just found to be
          odd, which was pointed to by the loop index.
      Set the corresponding element in M to one.
```

The indentation of part of item 3 is used to indicate the subject's knowledge that these actions are to be performed only if the test is satisfied.

In the program, this plan is represented as follows (Each numbered item is a separate element of the plan.):

```
1.  (CREATE (POINTER (NEXT ODD)) (BEGINNING (LIST OF NUMBERS)))
2.  (LOOP-THROUGH (LIST OF NUMBERS))
3.  (IF ((EVEN PARITY)
         (ARRAY-ELEMENT (LIST OF NUMBERS)
                        (VARIABLE (LOOP-INDEX))))
        (GOTO LOOP-END))
4.  (BEGIN! (NOT-EVEN-PARITY))
5.  (SWAP-AND-INCREMENT
     ((ARRAY-ELEMENT (LIST OF NUMBERS) (VARIABLE (LOOP-INDEX)))
      (ARRAY-ELEMENT (LIST OF NUMBERS) (POINTER (NEXT ODD))))
     (POINTER (NEXT ODD)))
6.  (SET (ARRAY-ELEMENT (AUXILLARY ARRAY) (POINTER (NEXT ODD)))
         1)
7.  (END! (NOT-EVEN-PARITY))
8.  (END! (LOOP-THROUGH))
```

The (BEGIN!) and (END!) elements have the same significance as does the indentation in the previous representation of the plan.

Interpretation of this representation is quite simple. The first word indicates the general action to be performed - SET, IF, SWAP, etc. Following it are the objects of the action with all the information belonging to an individual object grouped together within parentheses. Thus, (POINTER (NEXT ODD)) describes a pointer to the next odd number. Elements of arrays require two pieces of information, the name of the array and the name of the pointer indicating the specific element; for example, (ARRAY-ELEMENT (LIST OF NUMBERS) (VARIABLE (LOOP INDEX))) specifies an element of the array containing the list of numbers which is pointed to be a variable which is the loop index.

comparison between this representation of the plan and the informal one reveals that, despite syntactic differences, the two both contain the important features of the plan in this problem: both use an array as the primary data structure, two pointers to keep track of positions in the array, and iteration construct to step the pointers through the array.

Important to what is meant by a plan and by code generation in this model is what this plan omits which the code generation process supplies. First, though trivial, the plan has a different syntax and set of semantic conventions than does the final code. Second, the plan omits any operations which are necessary to create, label, and initialize data structures; in this case, the plan does not gives names to the array and the two pointers nor does it provide the DIMENSION declaration which is necessary to use a FORTRAN array. Third, the plan does not provide the mapping between the operations that are to be performed and the available constructs of the language; in this example, the plan provides no way of getting from the "swap" operation to the sequence of three assignment statements that are actually necessary to implement the operation in FORTRAN. Finally, the plan provides no information on how to perform whatever transformations may be necessary to go from the control structure of the plan to the control structure actually used in the program; in this case, it does not supply the information on how to go from the block structure used in the plan to the necessary sequence of FORTRAN GOTO statements. Supplying all the information that is omitted by the plan itself is the justification for the use of the complex problem-solving structure for code generation that was described in the previous chapter.

4.1.2 Operation of the Program for Lines 13-60

Lines 13-60 of the protocol consist of only code generation, and they have been simulated using the model. A complete trace of the model's behavior is given in Section 4 of Appendix 5. At the beginning of the segment, it is presumed that the subject has available the plan just described. In addition, he already has some information about two of the data structures he will use in the program, the two arrays, L and M. This information was acquired from the problem instructions read by the subject while doing problem HENRY, the first problem in the set; it's retention for application in this problem can be inferred from its use throughout the program. In the program, this knowledge is presumed to be stored in the MEANINGS structure in the following form:

1. (MEANS (LIST OF NUMBERS)), (TYPE ARRAY), (LENGTH 100), (NAME L)
2. (MEANS (AUXILLARY ARRAY)), (TYPE ARRAY), (NAME M), (LENGTH 100))

The first line is the set of attribute-value pairs that describe L; the second line is the set for M. They indicate that the first quantity MEANS LIST OF NUMBERS, that it is of TYPE ARRAY, LENGTH 100, and is called L , in the program. and that the second quantity MEANS AUXILLARY ARRAY, is of TYPE ARRAY, is called M in the program, and is of LENGTH 100.

Plan element 1 specifies the creation of a variable which is a pointer to the next odd number and which has an initial value equal to the beginning position of the list of numbers. When this plan element is followed, it has two major effects: the creation of an entry in meanings for the pointer and the generation of code to initialize the pointer to 1. The generation of this code by the subject may be seen in lines 13-23 of the protocol:

S13:SO, POINTER ONE
S14:POINTER ONE IS A POINTER TO

4

A15:[WRITES PTR1]
S16:IT'S A POINTER FOR NEXT ODD
S17:NEXT ODD
S18:AND THE OTHER ONE IS JUST GOING TO GO THROUGH THE ARRAY SO
S19:I'LL JUST WRITE THIS NEXT ODD
S20:START OUT AND
A21:[WRITES NEXTODD=1]
S22:POSITION ONE LET'S JUST SAY
S23:DO 10 I EQUALS

(Note the change in the name the subject gives to the pointer between lines 13-15 and the remainder of the segment; as was mentioned previously, no attempt was made by the program to model this behavior.)

When the system begins operation to produce this same code, STM is presumed to have the following contents:

1. (PLAN-ELEMENT (CREATE (POINTER (NEXT ODD)) (BEGINNING
   (LIST OF NUMBERS))))
2. (CODE)
3. (MEANINGS)
4. EMPTY
5. EMPTY
6. EMPTY
7. EMPTY
8. EMPTY
9. EMPTY
10. EMPTY
11. EMPTY
12. EMPTY
13. EMPTY
14. EMPTY

"EMPTY" is used to indicate STM slots containing material irrelevant to the programming task.

The first element of STM is the first element of the plan. The next two elements are pointers to the CODE and MEANINGS structures respectively.

The first production to be invoked is PLAN-CREATE-POINTER-1. Its invoking condition is the presence of an element in STM which matches the pattern, (PLAN (CREATE (POINTER *REST*1*) *REST*2*)). A verbal statement of this pattern is "any PLAN which calls for creating a pointer described by what matches *REST*1* and initializing it to whatever matches *REST*2*."

The production has three actions. The first of these is to use the function, NEWMEANING, to add the following group to MEANINGS:

(MEANS (NEXT ODD)), (TYPE POINTER), (NAME NEXTODD)

Since no second argument is supplied, this entire list is added to MEANINGS as a single group.

The second and third actions of this production begin the generation of code to initialize the pointer. The second action has two parts: first, it takes a template for setting two things equal and uses PUTARG to instantiate it by plugging in (POINTER (NEXT ODD)) and (BEGINNING (LIST OF NUMBERS)) as parameters. Before this instantiation the template looks as follows:

(*GETARG* 1) = (*GETARG* 2) (NEWLINE)

After instantiation it appears as:

(POINTER (NEXT ODD)) = (BEGINNING (LIST OF NUMBERS)) (NEWLINE)

Once it is instantiated, the second part of the action is to call NEWMEANING to add the template along with its effect to MEANINGS to give the following group in MEANINGS:

```
(EXPRESSION ((POINTER (NEXT ODD)) = (BEGINNING (LIST OF NUMBERS))
      (NEWLINE)))

(EFFECT ( EQUAL (POINTER (NEXT ODD)) (BEGINNING
      (LIST OF NUMBERS))))
```

The final action of the production is to apply the code generation function, CODE-GEN, to the instantiated template in order to produce the actual code. As described previously, CODE-GEN proceeds by passing the elements in the template one at a time to CODE-EL. The first element in the template is (POINTER (NEXT ODD)). Since this is not an actual code element, CODE-GEN calls GETMEANING, with arguments of (TYPE POINTER) and (MEANS (NEXT ODD)), to search for a name for the item in MEANINGS; since the quantity has already been created, it finds an entry for it, and returns the name, NEXTODD. CODE-EL passes this back to CODE-GEN, which adds it to the buffer of code being created. The next element in the template is "=". Since this is a piece of actual code, it is returned by CODE-EL unchanged and is added directly to the buffer for code being created.

The third element in the template is (BEGINNING (LIST OF NUMBERS)). As in the case of the first element, CODE-EL calls GETMEANING to find whether the requested quantity is available from MEANINGS. Since it is not, CODE-EL reports back failure (NIL). CODE-GEN responds to this failure by creating two new STM elements and then returning.

These new elements are

```
                    (NEW-CODE (NEXTODD =))
        (CODE-GENERATION TEMPLATE-1 POSITION-1 SIGNALS-1)
```

The first of these contains all the new code that was in CODE-GEN's buffer; the second contains the status of CODE-GEN at this point and corresponds to a node in a goal tree.

The next production that is invoked is NEW-CODE-3. Its invoking conditions are the presence of elements matching the patterns:

```
(NEW-CODE *ANY*)
(PLAN-ELEMENT *REST*)
(MEANINGS)
(CODE)
```

In addition to rehearsing the PLAN-element, CODE, and MEANINGS, it has two other actions. The first is to call the ADDCODE function on the new code contained in the NEW-CODE element; this function adds the new code onto the CODE structure. The second action is to call the REPLACE function, described in the section on STM structure, to change the word, "NEW-CODE," in the first element to "OLD-CODE."

When this production is completed, the CODE-GENERATION element left in STM by the previous production, the CODE and the MEANINGS elements, and the absence of any GOAL elements together serve as the invoking conditions for the production, CONTINUE-CODE-GENERATION-1. This production marks the CODE-GENERATION element as an OLD-CODE-GENERATION and calls CODE-GEN with the remainder of the template at the point at which coding was interrupted. This time when CODE-GEN encounters the (BEGINNING (LIST OF NUMBERS)) element it creates the new STM elements:

(GOAL (BEGINNING (LIST OF NUMBERS)))

and

(CODE-GENERATION TEMPLATE-2 POSITION-2 SIGNALS-2)

At this point, STM appears as:

1. (CODE-GENERATION TEMPLATE-2 POSITION-2 SIGNALS-2)
2. (GOAL (BEGINNING (LIST OF NUMBERS)))
3. (CODE)
4. (MEANINGS)
5. (PLAN-ELEMENT (CREATE (POINTER (NEXT ODD)) (BEGINNING
   (LIST OF NUMBERS))))
6. (WRITTEN-CODE (NEXTODD =))
7. (OLD-CODE-GENERATION TEMPLATE-1 POSITION-1 SIGNALS-1)
8. EMPTY
9. EMPTY
10. EMPTY
11. EMPTY
12. EMPTY
13. EMPTY
14. EMPTY

Note the presence of the CODE-GENERATION and OLD-CODE-GENERATION elements which perform the equivalent of a goal tree for code generation by indicating that coding of a template is still incomplete.

A production, GOAL-BEGINNING-1, has as its invoking condition the pattern, (GOAL (BEGINNING *REST*)), which is matched by the GOAL element in STM. It has two main actions. The first is to call NEW-MEANING with (BEGINNING 1) as its first argument and (MEANS (LIST OF NUMBERS)) as the second; the use of the second argument has the effect of causing the first attribute-value pair to be added to the same group as the second pair or set, rather than creating a new set. This means that the information that the array begins at 1 is added to the information in MEANINGS about the array, L. The entire entry in MEANINGS now is:

```
(MEANS (LIST OF NUMBERS))
(TYPE ARRAY)
(LENGTH 100)
(NAME L)
(BEGINNING 1)
```

The second action is to add the element, (NEW-CODE 1), to STM. The conditions for invoking NEW-CODE-3 are then met; it adds the 1 to the CODE structure.

Once this GOAL is satisfied, encoding of the template is completed, and the effect associated with this code is retrieved by CODE-GEN (using the function, GETMEANING) and placed into STM as an EFFECT element. STM has the following appearance after all the code that has been generated is written out into the CODE EM:

```
1. (CODE)
2. (PLAN-ELEMENT (CREATE (POINTER (NEXT ODD)) (BEGINNING
      (LIST OF NUMBERS))))
3. (WRITTEN-CODE (CRLF))
4. (EFFECT (EQUAL (POINTER (NEXT ODD)) (BEGINNING
      (LIST OF NUMBERS))))
5. (MEANINGS)
6. (WRITTEN-CODE 1)
7. (OLD-GOAL (BEGINNING (LIST OF NUMBERS)))
```

8. (OLD-CODE-GENERATION TEMPLATE-2 POSITION-2 SIGNALS-2)
9. (WRITTEN-CODE (NEXTODD =))
10. (OLD-CODE-GENERATION TEMPLATE-1 POSITION-1 SIGNALS-1)
11. EMPTY
12. EMPTY
13. EMPTY
14. EMPTY

CODE now contains:

```
NEXTODD = 1
```

The production that will be evoked under these conditions, EQUIVALENT-EFFECT-1, calls the COMPARE-EFFECT function to determine if the effect of the code is the one desired for the current plan element. Since it is, the production system will place the next plan element into STM.

Summarizing the operations performed by this sequence, it creates the line of code, NEXTODD=1, and assigns it an effect. In the course of doing so, two goals are created and satisfied, one for the variable name, and one for its initial value. Comparing this with the subject's behavior in lines 16-21, the same basic elements are seen. Line 16 contains the subject's generation of the name; line 22 reveals generation of the initial value from information about the "list of numbers" array; and, line 21 shows the subject's generation of the code itself.

The next plan element calls for looping through the list of numbers. The corresponding segment in the protocol is:

S23:DO 10 I EQUALS
S24:LET'S MAKE THIS A 20
A25:[WRITES DO 20 I=1,100]
S26:I EQUALS ONE TO 100

Coding of this plan element follows the same basic outline as for the first one. Two goals are generated and satisfied, one for a label for the end of the loop and one for the loop index. The code that is generatec is:

```
DO 10 I=1, 100
```

In the protocol, the subject first writes this code the same way the program does, but then changes the label to 20. The change does not appear to be motivated by the achieved or desired effect of code, but, rather, by the subject's naming conventions. Since, as discussed in the section on NEWQUAN, the model does not include this naming behavior, the subject's label alteration is not followed by the model.

As with the previous template, completion of coding results in the placement of the effect of the code into STM. Additionally, in this case, an element of the following form is placed in STM:

```
(CODE-CONDITION (OPEN-DO-LOOP (LABEL (LOOP-THROUGH
      (LIST OF NUMBERS)(LOOP-THROUGH(LIST OF NUMBERS)))))
```

It represents the programmer's knowledge, shown in lines 50-51 of the protocol, that the DO loop he has just created is still open.

If this element were not rehearsed, it would eventually be driven off the end of STM; this would correspond to the error situation in which the programmer forgets to terminate a loop. In this case, the LOOK-AT-CODE mechanism described previously is used to prevent this situation from occuring by assuming that the item is rehearsed whenever the programmer sees the DO statement in the written code.

Following the plan element for looping is one that calls for performing a test operation on the element of the array pointed to by the loop index. Coding begins by invoking PLAN-IF-1 which responds to STM elements matching the patterns, (PLAN-ELEMENT (IF *REST*)) and (OLD-PLAN-ELEMENT (LOOP-THROUGH *LIST*)). This production has three actions. First, it rehearses the element which matched the first pattern. Second, it calls NEW-MEANING to place the instantiated template, IF-1, and its effect in to MEANINGS. (The instantiation is accomplished as part of the argument to NEW-MEANING. Third, the production calls CODE-GEN with the instantiated template as its argument.

Coding of this template proceeds in the same general manner as has been described previously. Noteworthy is the creation of a goal, (GOAL (TEST ((EVEN PARITY)(ARRAY-ELEMENT (LIST OF NUMBERS) (VARIABLE (LOOP-INDEX)))))), for the test inside the IF statement. This goal matches the pattern, (GOAL (TEST ((EVEN PARITY) *REST*))) and invokes the production, GOAL-TEST-EVEN-PARITY, which creates the code for the test. Coding of the remainder of the IF statement then follows. When this is complete, the plan element is also completed.

The equivalent behavior on the part of the subject appears in lines 27-32 of the protocol:

S27:IF THE THING IS ODD -IF
S28:L SUB I DIVIDED BY TWO TIMES TWO EQUALS L SUB I
S29:THEN WE WANNA SAY
S30:GO TO 20
A31:[WRITES IF(L(I)/2*2=L(I))GO TO 20]
S32:IF IT'S NOT ODD

Note that generation of the test for the IF statement appears as a separate line in the protocol because the subject breaks off a previous phrase before stating the code for it. This suggests that the subject also treats creation of the test as a separate goal from the rest of the IF statement and that the model accurately reflects his behavior. (This separation appears even more clearly in two of the other segments modeled by the program.)

After completion of the plan element for the test, the next element in the plan is a special BEGIN! marker; it represents the subject's knowledge, as seen in lines 33 and 34 of the protocol, that the following calculations are a group that is performed only if the number is odd.

S33:SO IT IS ODD
S34:AND IF IT IS ODD

No code is generated from it, but it does cause the rehearsal of any CODE-CONDITION elements which are in STM.

Within the group that the BEGIN! marker initiates, there are two other plan elements. The action of the first of these is represented as SWAP-AND-INCREMENT. Though two distinct functions are named, they are represented as a single, compound action since the subject appears to treat them this way; in lines 35-47 of the protocol, generation of the code for the two functions is intermixed:

S35:THEN WE'LL SAY SWITCH IT WITH NEXT ODD
S36:PUT A ZERO HERE
A37:[CHANGES NEXTODD=1 TO NEXTODD=0]
S38:AND WE'LL SAY K EQUALS L SUB I
A39:[WRITES K=L(I)]
S40:L SUB I EQUALS
S41:L SUB NEXT ODD
A42:[WRITES L(I) =L(NEXTODD)]
S43:L SUB NEXT ODD - NO NEXT OOD
S44:NEXT ODD EQUALS NEXT ODD PLUS ONE
A45:[INSERTS NEXTODD=NEXTODD+1 BEFORE K=L(I)]
S46:NEXT ODD L EQUAL K
A47:[WRITES L(NEXTODD)=K]

In order for the SWAP-AND-INCREMENT operation to be performed properly, a necessary prerequisite is that NEXTODD have the proper initial value. In other situations, this value might be determined by first writing the code for the operation and then symbolically executing it for the case in which the first two elements of the array are to be swapped; in this case, however, the subject apparently is aware, before the rest of the code is written, that NEXTODD should initially be 0. He therefore goes back to the beginning of the program where he has written NEXTODD = 1 and re-writes it as NEXTODD = 0 (in lines 36-37 of the protocol).

This ability to re-write incorrect code is provided for within the program; every time CODE-GEN is called with a new template, it uses a function, SIMILAR-EFFECT, to check whether the effect of the new template matches the effect of a previous template stored in MEANINGS in certain ways. If it does, then is assumed that the code generated from the new template is intended to re-write the older code. In this case, the two effects are:

```
old: (INITIALIZED ((POINTER (NEXT ODD))(BEGINNING
              (LIST OF NUMBERS))))

new: (INITIALIZED ((POINTER (NEXT ODD)) 0))
```

If the effects do match, then CODE-GEN places a pointer to the older code into STM in the form of an element, (OLDCODE). As each piece of code from the new template is placed into STM, a production, NEW-CODE-1, uses the function, HOW-MODIFY, to determine how the new code is to modify the older code, whether it should replace it, be inserted before it, or be inserted after it. One of the 3 productions, REPLACE-CODE-1, INSERT-BEFORE-1, and ADD-ON-CODE-1, then makes the modification and updates the OLDCODE pointer. In this problem, CODE-GEN first produces (NEWCODE (NEXTODD = 0)) and (OLDCODE). NEW-CODE-1 calls HOW-MODIFY and changes the first element to (REPLACE (NEXTODD = 1 CRLF) (NEXTODD = 0 CRLF)). (The "CRLF" stands for "carriage return - line feed " and is used to indicate the end of a line in the code.)

Once NEXTODD has been properly initialized, the subject begins to write the code for interchanging the two elements of the arrays, in lines 38-47. In the course of doing so, he discovers that he has not incremented NEXTODD, and he interrupts writing the code for swapping the elements to insert NEXTODD = NEXTODD + 1 before the swap. There is no evidence in the protocol that his discovery was the result of attempting to execute the code for some particular value of NEXTODD. Instead, it seems to have come about as a result of his using the name, NEXTODD, which served to remind him that he had not yet written the increment statement.

Since the behavior comes about as a consequence of seeing something in the written code, the program uses the LOOK-AT-CODE mechanism to simulate this behavior, but because the LOOK-AT-CODE element is placed into STM only when the entire template is completed, the simulation is not exact. As soon as the code for the entire swap of the array elements has been written out into CODE, an element is introduced into STM via the LOOK-AT-CODE device which indicates that NEXTODD has a value of 0 and cannot be used as a subscript. This condition causes a production, REPLACE-CODE-1, to be invoked which uses the code re-writing mechanism described previously to insert the code. Thus, in the program, the modification to the initialization of NEXTODD takes place after, not during, the writing of the swap code.

At this point, the code appears as

```
NEXTODD = 0
DO 100 I=1,100
IF(L(I).EQ.L(I)/2 * 2) GOTO 10
NEXTODD = NEXTODD + 1
K = L(I)
L(I) = L(NEXTODD)
L(NEXTODD) = K
```

After the SWAP-AND-INCREMENT has been completed, the next plan element calls for setting the corresponding element in the M array to one. The subject does this as a single operation with no evidence of creation of a new variable name or other subgoals:

```
S48:AND M SUB NEXT ODD EQUALS ONE
A49:[WRITES M(NEXTODD)=1]
```

Similarly, in the program, the creation of code for it is also very straight-forward. It is done completely by PLAN-SET-EQUAL-1 which encodes the template, EQUAL-1. No other subgoals or productions are involved.

The final element of the plan is the END! that closes the block of code following the IF statement. A production, PLAN-END-CODE-CONDITION-2, sensitive to the conjunction of an END! element and a CODE-CONDITION of OPEN-DO-LOOP, then creates the CONTINUE statement which closes the loop and completes the program. The corresponding behavior on the subject's part is:

```
S50:20 CONTINUE
A51:[WRITES 20 CONTINUE]
```

The final program is:

```
NEXTODD = 1
DO 10 I = 1 , 100
IF ( L ( I ) / 2 * 2 . EQ . L ( I ) ) GOTO 10
NEXTODD = 0
NEXTODD = NEXTODD + 1
K = L ( I )
L ( I ) = L ( I )
L ( I ) = K
M ( NEXTODD ) = 1
10 CONTINUE
```

It differs from the subject's code only in the label names.

In gaining an overview of the relationship between the model trace and the protocol, it is important to note the level at which comparison is appropriate. No attempt was made to achieve correspondence at the level of a production for each word or phrase; instead, the model is intended to reproduce just two characteristics of the coding behavior: (1) the overall order in which code is created and (2) the size of the units in which it is created, defined in terms of what is accomplished by single operations. From this perspective, the model fits the protocol reasonably well. With the exception of the problem in incrementing NEXTODD caused by the LOOK-AT-CODE mechanism, the order of code generation by both the subject and the model match. Additionally, the unit sizes also match well. Both the subject and the model create the IF statement as two distinct pieces of code while the swap operation is a single piece even though it takes several lines in the program. Both also store and retrieve some of the same information about code quantities since both set the corresponding element in M without creating new goals for information about variable names. Thus, when compared at the level of code generation order and unit size, the model adequately reproduces the important characteristics of the subject's behavior.

4.2 Problem LEE

Problem LEE was the 19th problem done by the subject; the complete protocol is given in Appendix 2. It lies below the median in writing time but it took a longer than average time to debug and run. The task was to find all sets of numbers such that all members of the set were multiples of the smallest member in the set, excluding one and zero from the sets. The sets were then to be placed at the beginning of the array, and the corresponding positions in the M array for the first set were to be set to one, the second set to two, etc.

This problem was selected for modeling because, for the theory, it is almost a classical one; not only are all three of the processes of the theory explicitly visible in the protocol, but considerable interaction among the processes may also be seen. The three processes occurred in the following order:

lines	1-51	Understanding
	52-66	Planning
	67-140	Coding
	141-179	Planning
	180-387	Coding

As can be seen, the Understanding process at the beginning of the protocol is a quite protracted one. The main issue which consumes this effort is how to handle negative numbers. If the problem is interpreted so that their actual, rather than absolute, value is used, and "multiples" is interpreted as meaning "multiplied by a positive number," then each negative number would form a set by itself. Since this seems peculiar to the programmer, he attempts to verify that the interpretation is a correct one by using information from a supplementary resource, asking the experimenter. The experimenter leaves the interpretation up to the programmer; the programmer then decides to attempt to write the program using the interpretation in which the actual values of negative numbers are used.

Parenthetically, attention should be drawn to the clear design-task characteristics of the understanding process in this case. The basis on the which the interpretation of the problem is questioned is the the programmer's notion of what constitutes a suitable problem; it does not seem reasonable to him that a correct interpretation of the problem would result in many of the sets of multiples having only one member. A test for what constitutes a "suitable" problem requires access to a wide range of diverse information. The use of this "suitable" problem idea in the understanding process guarantees the main characteritic of design tasks, the continuous introduction of new knowledge during the course of problem solving.

After the interpretation issue is clarified, an initial plan is created; as in problem RICHARD, creation takes place rapidly with no evidence of extended problem-solving activity. This initial creation is visible in lines 52-66 of the protocol; the plan conceived in this segment is apparently the one used until the Planning segment that begins in line 141. From these lines alone, the plan consists of first ordering the numbers and then going through and finding the multiples. From lines 94-95 in the Coding section, a more elaborate statement of the plan can be inferred: First find a positive. If any negatives are found, the corresponding positions in M are to be set to 1, as is seen from lines 108-111; and, if the loop terminates, then the program is done operating (from lines 112-133). When a positive number is found, the plan is to check whether its successors are positive; if they are, they are to be swapped with any non-multiples which lies between the positive number and the multiple that has been found (from lines 119-124 and 132-139 and 143 in the next planning section).

The complete plan at the end of this first Planning section looks like:

1. Order the numbers.

2. Go through the numbers and find a positive one.

   For each negative number found, set the corresponding position in M to the value of the loop index.

   If no positive numbers are found, then the program is done.

3. Test the rest of the numbers in the array to find whether they are divisible by the positive number that has been found.

4. Keep a pointer to the first non-positive after the positive has been found.

5. Each time a multiple is found, swap it with the element that the pointer points to and update the pointer.

Again, the indentation is used to indicate that operations are to be performed only if a condition holds.

The program representation of this plan is:

```
1. (ORDER (LIST OF NUMBERS))
2. (FIND-EXISTENCE ((FIRST POSITIVE) (LIST OF NUMBERS)))
3. (BEGIN! (OTHERWISE))
4. (SET (CORRESPONDING-ELEMENT (AUXILLARY ARRAY)
                               (VARIABLE (LOOP-INDEX)))
        (VARIABLE (LOOP-INDEX)))
5. (END! (OTHERWISE))
6. (END! (FIND-EXISTENCE-LOOP-THROUGH))
7. (BEGIN! (POSITIVE-FOUND))
8. (FIND-AND-SWAP
          ((MULTIPLE) (LIST OF NUMBERS) (VARIABLE (LOOP-INDEX)))
          ((LIST OF NUMBERS) (VARIABLE (INNER-LOOP-INDEX))))
```

Note that the END! in item 6 does not have a corresponding BEGIN!. This is because the FIND-EXISTENCE function in item 2 belongs to a class of items which open DO loops and, therefore, act as implicit BEGIN!s. (For a full list of this class, see Appendix 4.) The BEGIN! in item 7 is unbalanced because the plan at this point is still incompletely specified; the subject has still not decided what to do when all the multiples of the number have been found. (As in the preceding problem, this plan is still quite far from the eventual program. In particular, the plan uses a different set of operators and operations than those available in FORTRAN, and it omits the naming, creating, and initializing of data structures.)

This plan as originally conceived will not work; interchanging elements which are multiples with elements which are not destroys the ordering of the array. The subject discovers this in the course of writing the code for doing the interchange, in lines 129-145 of the protocol. How this discovery is made is not quite clear from the protocol, since the subject writes very little of the code before making the discovery; however, it is probably a reasonable inference that he partially generates and symoblically executes the code without writing or vocalizing it. The symbolic execution reveals that, after finding the first set of multiples, the array will no longer be in increasing order and that he can no longer be sure that the next divisor he tries will be the smallest member in the set.

Since the problem in this case lies in the plan and not just in the code that has been generated to fulfill the plan, the planning processes is again invoked.; this is visible in lines 146-179 in the protocol. His starting point in modifying the plan is an idea that first appears in line 142 of the protocol: each time a multiple is found it is put in position at the beginning of the array, and all the non-multiples which preceeded it in the array are moved up one place to preserve the ordering of the array. He uses a diagram to verify and elaborate this idea, and, at line 180, begins using it to generate code.

Lines 180-302 are basically concerned with writing the code for finding the multiples and shifting them to the head of the array. Beginning about line 303 he begins to concern himself with terminating conditions. To generate the code for these, he again symbolically executes the code that he has already written, this time using various terminating conditions as the values for the execution; for example, in lines 303-320 he symbolically executes a branch condition with values of greater than 100 for the variable, NEXT. Once he has satisfied himself that the program works correctly for all these terminating conditions, he is done with writing the code for the program.

4.2.1 Program Operation for Lines 52-141

The program has been set up to operate for lines 52-141 of this protocol. This segment begins just after the initial plan creation and ends with the discovery that the original plan is inadequate.

The first element of the plan requires the ordering of the list of numbers. It is:

```
(PLAN-ELEMENT (ORDER (LIST OF NUMBERS)))
```

Compared to the other elements in the plan, it seems to be at a higher levei of abstraction; however, there is no evidence in the protocol that it ever gets broken down any further, and code is generated directly from it. A possible explanation is that the subject has written the code for ordering numbers so often that it has become a planning primitive for him.

In the model, this code generation is handled as a series of steps in which the effect of one step, in combination with the plan, is used as the invoking condition for the next step. The first of these steps is represented by:

```
(GOAL (LOOP-THROUGH ((LIST OF NUMBERS)(ORDER (LIST OF NUMBERS)))))
```

which is interpreted as "goal for looping through the list of numbers as part of ordering the list of numbers." In response to this goal, a production is fired off which invokes the CODE-GEN function with the template, LOOP-1. When coding of this template is completed, it produces

```
DO 10 I=1,100
```

and leaves as its effect,

```
(EFFECT (LOOPED-THROUGH ((LIST OF NUMBERS)((ORDER (LIST OF NUMBERS))))
 (LABEL (LOOP-THROUGH ((LIST OF NUMBERS)((ORDER (LIST OF NUMBERS)))))))
```

This is interpreted as "the effect is to loop through the list of numbers to order the list of numbers, and the effect is complete at the label for looping through the list of numbers to order them." The corresponding behavior by the subject is:

S67:DO 100 I EQUALS ONE
S68:TO 100
A69:[WRITES DO 100 I=1,100]

The presence of this effect and the plan element in STM are invoking conditions for a production which produces:

```
    (GOAL (INNER-LOOP-THROUGH
            ((LIST OF NUMBERS)(ORDER (LIST OF NUMBERS))))))
```

This is a goal for the production of an inner loop that goes through the list of numbers. In turn, this results in a call to CODE-GEN with the template, LOOP-2. The code generated from it is:

```
DO 10 J=1,100
```

This has as its effect,

```
    (EFFECT (INNER-LOOPED-THROUGH
                ((LIST OF NUMBERS)((ORDER (LIST OF NUMBERS))))
                 (LABEL (LOOP-THROUGH (LIST OF NUMBERS))
                 (LENGTH (LIST OF NUMBERS))))
```

In the protocol this corresponds to:

S70:DO
S71:J EQUALS I PLUS ONE
S72:100
A73:[WRITES DO 100 J=1,100]

While this is the correct effect for the goal, it is not the correct effect for the plan. To order the numbers efficiently, the inner loop should begin at one past the current position of the outer loop and should run to one less than the outer loop. In the program, the subject sees this and corrects it in lines 74-78 of the protocol:

S74:OR I 1
A75:[WRITES ABOVE SECOND DO, I1=I+1]
S76:J EQUALS I 1
S77:99
A78:[ALTERS SECOND DO STATEMENT TO READ, DO 100 J=I1,99]

In the program, the correction is accomplished by a production, PLAN-ORDER-AFTER-WRONG-INNER-LOOP-1, which has as its invoking conditions, the presence of a plan element for ordering the list of numbers, the effect just given, and the absence of of an effect of incrementing a pointer. The latter condition is equivalent to specifying that the inner loop begins at one rather than beginning at a pointer which has a value one greater than the index of the outer loop. This production has two main actions. One of them is to use the OLDCODE mechanism to mark the DO loop that was just written as erroneous code which is to be replaced by new code. The second action is to introduce into STM a goal for creating a pointer which would be equal to the value of the outer loop index plus one. This goal appears as:

```
(GOAL (INCREMENT (INNER-LOOP-ORIGIN) 1)))
```

This has the effect of creating a variable which MEANS INNER-LOOP-ORIGIN and then creating the code to increment it by one.

The effect of this code and the plan element then serve as conditions for a production, GOAL-LOOP-THROUGH-1, which creates a DO loop that begins at this pointer and has the length of the list of numbers minus one as its upper bound. This goal appears as:

```
(GOAL (LOOP-THROUGH
      ((LIST OF NUMBERS)
      NIL
      (MINUS (LENGTH (LIST OF NUMBERS))1))))
```

This is interpreted as "goal for creating an inner loop through the list of numbers beginning at the default location (represented by the NIL) and running to the length of the list of numbers minus one." This goal results in a call to CODE-GEN using the template, LOOP-3. Since the OLDCODE element is present in STM, the function, HOW-

MODIFY, is called for each piece of code generated from this template. It marks these new pieces as replacements for the old DO loop. When coding of the entire template is completed, the combined result of the goal and the one previous to it is to produce the code, which performs the inner loop for the ordering operation:

```
I1 = I + 1
DO 10 I1= 1,99
```

In the protocol, up to line 97, the subject uses 100 as the terminating label for the DO loop while the program uses 10. The subject's change to 10 in lines 97-98 does not appear to be motivated by the achieved or desired effects of the code he is writing, but, as in a similar case in problem RICHARD, by his set of conventions for generating labels. As such, no attempt has been made to make the model produce the same behavior as the subject.

Once the two loops have been created, the next step in ordering the the numbers is to test whether the current largest number is larger than the next number to be tested in the array. This takes place in 80-83 of the protocol:

```
S80:IF L SUB I IS
S81:LESS THAN EQUAL TO
S82:L SUB J GO TO 100
A83:[WRITES IF(L(I).LE.L(J))GO TO 100]
```

The effect of completing the two DO statements serves as part of the invoking conditions for a production which creates a goal for writing this code. This goal appears as

```
(GOAL (IF (TEST (GREATER)
              (ARRAY-ELEMENT (LIST OF NUMBERS)
                             (VARIABLE (LOOP-INDEX)))
              (ARRAY-ELEMENT (LIST OF NUMBERS)
                             (VARIABLE (INNER-LOOP-INDEX))))
          (GOTO (LABEL
                 (LOOP-THROUGH ((LIST OF NUMBERS)
                                (ORDER (LIST OF NUMBERS))))))
```

This goal invokes the production, GOAL-IF-TEST-GREATER-1, which calls CODE-GEN with template, IF-1. IF-1 is a general template for IF statements which have a GOTO as their action. Code for the specific test within the IF statement is generated by creating a sub-goal for that specific test. The goal for creating the test for which element is greater appears as

```
(GOAL (TEST ((GREATER)
          (ARRAY-ELEMENT (LIST OF NUMBERS)
                   (VARIABLE (LOOP-INDEX)))
          (ARRAY-ELEMENT (LIST OF NUMBERS)
                   (VARIABLE (INNER-LOOP-INDEX))))))
```

The production, GOAL-TEST-GREATER-1, satisfies this goal by calling CODE-GEN with the template, TEST-2; it results in the code,

```
L(I).GT.L(J)
```

and has the effect,

```
(EFFECT (TEST ((GREATER)
          (ARRAY-ELEMENT (LIST OF NUMBERS)
                   (VARIABLE (LOOP-INDEX)))
          (ARRAY-ELEMENT (LIST OF NUMBERS)
                   (VARIABLE (INNER-LOOP-INDEX)))))
```

When the goal for the entire IF statement is satisfied, it produces the code,

```
IF (L(I).GT.(L(J)) GOTO 10
```

which has the effect,

```
(EFFECT (BRANCH-IF
    (GREATER
      (ARRAY-ELEMENT (LIST OF NUMBERS)
               (VARIABLE (LOOP-INDEX)))
      (ARRAY-ELEMENT (LIST OF NUMBERS)
               (VARIABLE (INNER-LOOP-INDEX))))
    (GOTO (LABEL
         (LOOP-THROUGH
          ((LIST OF NUMBERS)
           (ORDER (LIST OF NUMBERS))))))))
```

This is interpreted as "if the array element pointed to by the outer loop index is greater than the array element pointed to by the inner loop index, branch to the label which is the end of the loop for ordering the list of numbers."

If the current element is greater than the largest element found so far, then the next step in the ordering operation is to swap the two elements. In the program, the goal for doing the entire swap is satisfied by calling PLAN-ORDER-AFTER-BRANCH-IF-1 which calls CODE-GEN with a template, SWAP-1. This template, when completed, generates the following code for the swap:

```
LL=L(I)
L(I)=L(J)
L(J)=LL
```

This use of a single template to generate several lines of code is equivalent to the assertion that, for the subject, these several lines of code are a single knowledge unit. Lines 84-88 of the protocol support this assertion:

S84:OTHERWISE WE SAY L SUB I
A85:[WRITES LL=L(I)]
S86:L SUB I L SUB J
A87:[WRITES L(I)=L(J)]
S88:L SUB J EQUAL L

They show that the generation of this code takes place as a single, uninterrupted operation, which would be the case if knowledge about how to generate it were a single unit.

After the swap is complete, the code must be written to close the loop. It is done in lines 91 and 92 of the protocol:

A91:[WRITES 100 CONTINUE]
S92:ALL RIGHT, THEY'RE NOW IN ORDER

The goal for this is:

```
(GOAL  (LOOP-END
         (LABEL (LOOP-THROUGH ((LIST OF NUMBERS)
                              (ORDER (LIST OF NUMBERS)))))))
```

It is placed into STM as the result of a production, PLAN-ORDER-AFTER-SWAPPED-1, whose invoking conditions are the plan element, the effect of the swap, and the CODE-CONDITION element that was placed into STM when the loops were created. The goal is satisfied with a production, GOAL-LOOP-END-1, that calls CODE-GEN with the template, LOOP-END-1, to generate a CONTINUE statement. At this point, the completed code looks like:

```
DO 10 I = 1 , 100
I1 = I1 + 1
DO 10 J = 1 , 99
IF ( L ( I ) . GT . L ( J ) ) GOTO 10
K = L ( I )
L ( I ) = L ( J )
L ( J ) = K
10 CONTINUE
```

Completion of the loop also completes the plan element for ordering the list of numbers, as the subject states in line 92. The next plan element is

```
(PLAN-ELEMENT
   (FIND-EXISTENCE ((FIRST POSITIVE) (LIST OF NUMBERS))))
```

which calls for finding the first positive in the list of numbers, as can be inferred from lines 93-95 of the protocol:

S93:NOW WE SAY
S94:START FROM THE FIRST
S95:FIND THE FIRST POSITIVE

The first step in carrying out this plan element is to invoke PLAN-FIND-EXISTENCE-AFTER-FIRST-POSITIVE-1 which creates a goal,

```
(GOAL
   (FIND-EXISTENCE-LOOP-THROUGH
    (LIST OF NUMBERS)
    ((FIRST POSITIVE)
     ((FIND-EXISTENCE ((FIRST POSITIVE) (LIST OF NUMBERS)))))))
```

It specifies the creation of a loop for finding the existence of the first positive in the list of numbers; this is done by PLAN-FIND-EXISTENCE-FIRST-POSITIVE-1. This goal is satisfied by a production, GOAL-FIND-EXISTENCE-LOOP-THROUGH-1, which calls CODE-GEN with the template, LOOP-3, to create

```
    DO 20 I=1,100
```

The effect of this code serves as part of the invoking condition for the next step in finding the first positive, creation of an IF statement which tests whether the current element pointed to by the loop index is positive, as is done in lines 101-106 of the protocol:

```
S101:IF L SUB I IS GREATER THAN L
A102:[WRITES IF (L(I).GT. L(  ]
S103:IF IT'S GREATER THAN
S104:IT'S EITHER ZERO OR ONE
S105:ONE
S106:GO TO 30
```

The goal for this appears as:

```
  (GOAL
    (IF (TEST (POSITIVE)
              (ARRAY-ELEMENT (LIST OF NUMBERS)
                             (VARIABLE (LOOP-INDEX))))
        (GOTO (LABEL (POSITIVE-FOUND) SMALL))))
```

This is Interpreted as "goal for testing whether the element pointed to by the array index is positive; if it is, go to a label which means 'positive found'." Again, the general template, IF-1, is used in creating the code for satisfying this goal, so that the test itself appears as a separate goal. Note that in the protocol an indication of the validity of this separation appears in lines 101-106; the subject knows how to write the beginning of the IF statement and the GOTO at the end, but he puzzles over the test in the middle of the statement.

The goal for the test in the model is:

```
(GOAL (TEST ((POSITIVE)(ARRAY-ELEMENT (LIST OF NUMBERS)
                                      (VARIABLE (LOOP-INDEX))))))
```

Satisfaction of this goal permits the completion of the IF statement to give:

```
IF(L(I) .GT. 0) GOTO 30
```

The effect of this code serves to complete the plan element for finding the first positive in the list of numbers.

The next plan element is

```
(PLAN-ELEMENT (BEGIN! (OTHERWISE)))
```

The production that is sensitive to this plan element, PLAN-BEGIN-1, does not create any goals for code generation nor does it generate any code directly; instead, this plan element serves to indicate that what follows belongs together as one group inside the DO loop.

In the protocol, the subject indicates this by:

S108:OTHERWISE WE SAY 20 SAYS

The first plan element of this group is

```
(PLAN-ELEMENT
   (SET (CORRESPONDING-ELEMENT
         (AUXILLARY ARRAY)
         (VARIABLE (LOOP-INDEX)))
        (VARIABLE (LOOP-INDEX))))
```

which calls for setting the corresponding element of the auxillary array to the value of the variable which is the loop index. Since this plan element can be carried out by only a single line of code, the production which responds to this plan element, PLAN-SET-EQUAL-1, does not create any GOALs; instead, it calls CODE-GEN directly with the template, EQUAL-2, to produce the code,

```
    M(I)=I
```

In the protocol, this appears as:

```
S109:M SUB I EQUALS I
A110:[WRITES 20 M(I)=I]
S111: NUMBER
```

The suceeding plan element is:

```
    (PLAN-ELEMENT (END!))
```

This serves to match the preceding BEGIN! and indicates the end of the group of actions which are performed if the number is not positive. The production which responds to it and to the presence of a CODE-CONDTION element, PLAN-END-CODE-CONDITION-2, calls CODE-GEN with the template, LOOP-END-1, which produces the code,

```
    20 CONTINUE
```

to close the loop.

The END! plan element is followed by:

```
     (PLAN-ELEMENT (END! (FIND-EXISTENCE-LOOP-THROUGH)))
```

A production with patterns, (PLAN-ELEMENT (END! *REST*)) and (OLD-EFFECT (LOOP-TERMINATION *REST*)), matches this element and the element,

```
    (OLD-EFFECT
      . (LOOP-TERMINATION
         (FIND-EXISTENCE-LOOP-THROUGH
((LIST OF NUMBERS) ((FIRST POSITIVE)
                   ((FIND-EXISTENCE
                     ((FIRST POSITIVE) (LIST OF NUMBERS)))))))
```

The latter is the effect of the CONTINUE statement that terminated the loop. The production then generates:

```
GOTO 300
```

This is the branch that is taken if no positive numbers are found. The equivalent behavior is seen in lines 112-115 of the protocol:

```
S112:AND DOWN HERE IF IT EVER GETS THROUGH THE LOOP WE'RE DONE
S113:GO TO EXIT
S114:EXIT WE'LL CALL 500
A115:[WRITES GO TO 500]
```

Note the difference in label numbers between the program and the protocol.

The next plan element is:

```
(PLAN-ELEMENT (BEGIN! (POSITIVE-FOUND)))
```

As in the other uses of BEGIN! elements, this marks the beginning of a group or section of code, in this case, that for the action to be taken when the first positive is found. It results in a call to CODE-GEN with the template, LABEL-1, which generates the label, 30, corresponding to the subject's behavior in lines 116-118:

```
S116:OK, 30
A117:[WRITES LABEL 30]
S118:NOW THAT WE HAVE THE FIRST ONE
```

After this label is generated, the next plan element is

```
(PLAN-ELEMENT
  (FIND-AND-SWAP
   ((MULTIPLE) (LIST OF NUMBERS) (VARIABLE (LOOP-INDEX)))
   ((LIST OF NUMBERS) (VARIABLE (INNER-LOOP-INDEX)))))
```

which calls for finding multiples in the list of numbers and swapping them with non-multiples. This plan element is never completed, since, in the course of it, the subject discovers the error in his plan.

The code completed by the program up to this point is:

```
DO 10 I = 1 , 100
I1 = I1 + 1
DO 10 J = 1 , 99
IF ( L ( I ) . GT . L ( J ) ) GOTO 10
K = L ( I )
L ( I ) = L ( J )
L ( J ) = K
10 CONTINUE
DO 20 I = 1 , 100
IF(L(I) .GT. 0) GOTO 30
M ( I ) = I
20 CONTINUE
GOTO 300
30
```

As noted, it differs from the code written by the subject only in the label names. Again, the model trace creates code in about the same size of unit and in the same order as the subject does.

4.3 Problem JOHN

Problem JOHN, the fourth problem done by the subject, took 27.5 minutes for the code to be written (slightly above the average). In it, the subject was asked to write a program which would take every other number and place it at the beginning of the array and put the original position of the number in the corresponding position in M. (The complete protocol is given in Appendix 2.)

This problem was selected for modeling because, surprisingly, considering the relative simplicity of the problem, the protocol shows a very extended planning phase, taking a total of the first 263 lines of protocol (about 21 minutes worth). Of interest was whether this extended planning would have any effect on the coding phase.

Most of the planning is concerned with finding a way to solve the problem only by moving the elements around within the array, L, instead of copying them into another array. He begins by trying to work out the mappings from the initial positions to the final ones to develop a formula for the subscript numbers. Once he has done this, he continues to work with the subscript mappings and hits upon the idea that the mapping can be completely circular and that the problem may be solved simply by exchanging pairs of elements in the proper order. To test this idea, he begins to work out the entire set of mappings for the 100 array elements but gives up when he sees no regular pattern emerging. He then works out the mappings for a very short array using a diagram and discovers that simply exchanging elements pairwise will not work, at least in the order in which he is trying to do it. He then goes ahead and works out a solution involving copying the arrays.

The basic structure of this plan is seen in lines 269-276 and involves copying the array elements into M and then copying them back into L. That he intends to do the copying as two separate operations, one for the odd numbers and one for the even numbers, can be inferred from lines 289-290 and from the code he actually writes. Similarly, his intent to do the copying back in two parts is obtained from lines 332-335 and from the written code. The complete plan that he has when he begins to write cooe is:

1. Copy the elements at odd positions in the list of numbers into the first half of the auxillary array.
2. Copy the elements at even positions in the list of numbers into the second half of the auxillary array.
3. Copy the first half of the auxillary array into the first half of the main array and set the first half of the auxillary array to the numbers of the even positions.
4. Copy the second half of the auxillary array into the second half of the main array and set the second half of the auxillary array to the numbers of the odd positions.

In the program, this plan is represented as:

```
((COPY ((ODD-POSITIONS) (LIST OF NUMBERS))
      ((FIRST-HALF) (AUXILLARY ARRAY)))
 (COPY ((EVEN-POSITIONS) (LIST OF NUMBERS))
      ((SECOND-HALF) (AUXILLARY ARRAY)))
 (COPY-AND-SET
  (((FIRST-HALF) (AUXILLARY ARRAY))
   ((FIRST-HALF) (LIST OF NUMBERS)))
  ((AUXILLARY ARRAY) (EVEN-POSITIONS)))
 (COPY-AND-SET
  (((SECOND-HALF) (AUXILLARY ARRAY))
   ((SECOND-HALF) (LIST OF NUMBERS)))
  ((AUXILLARY ARRAY) (ODD-POSITIONS))))
```

Note that this plan is not a very efficient one, even for the basic method chosen; by using appropriate subscript expressions, it is possible to solve the problem using two loops instead of four. The subject is aware of this; in lines 325-334 he attempts to do the second two plan elements as one single loop, but changes his mind in line 334, perhaps because the effort involved in figuring out the proper subscripts is too large. In terms of the theory this corresponds to modifying or revising the first plan, attempting to code the revised plan, having the revised plan fail (too high an effort expenditure), and then returning to the original plan.

4.3.1 Model Operation for Lines 279-359

Lines 279-359 of the protocol have been modeled with the program; they cover the generation of code for the entire program.

At the beginning, STM contains

```
(PLAN-ELEMENT (COPY ((ODD-POSITIONS) (LIST OF NUMBERS))
      ((FIRST-HALF) (AUXILLARY ARRAY)))
```

which is the first element of the plan that the subject has for solving the problem. The production system responds to this with COPY-LOOP-THROUGH-1 which creates a goal for looping through the list of numbers:

```
(GOAL
   (LOOP-THROUGH
    ((LIST OF NUMBERS)
     (COPY ((ODD-POSITIONS)(LIST OF NUMBERS))
          ((FIRST-HALF)(AUXILLARY ARRAY)))))
     NIL
     (VALUE ((ODD-POSITIONS)(LIST OF NUMBERS)))))
```

This requires the creation of a loop that runs through the list of numbers (to copy the odd positions into the first half of the auxillary array) which starts at the default value of 1 (indicated by the NIL) and has as its upper bound the value of the number of odd positions in the list of numbers. The production, GOAL-LOOP-THROUGH-2, is invoked in response to this goal; it calls CODE-GEN with template, LOOP-2, to create

```
DO 10 I=1,100
```

corresponding to the code that the subject creates in lines 279-282:

```
S279:DO 10
S280:I GOES EQUALS ONE TO
A281:[WRITES DO 10 I=1, ]
S282:LET'S WORRY ABOUT WHERE IT'S GONNA GO TO LATER
```

The production also leaves a code condion element in STM,

```
(CODE-CONDITION
   OPEN-DO-LOOP
   (LABEL
    ((LIST OF NUMBERS)
     ((COPY ((ODD-POSITIONS) (LIST OF NUMBERS))
           ((FIRST-HALF) (AUXILLARY ARRAY)))))))
```

which indicates that the DO loop has not been terminated yet.

The effect of this code and the plan element are the invoking conditions for the production, COPY-SET-EQUAL-1, which creates the goal:

```
(GOAL
   (SET (ARRAY-ELEMENT (AUXILLARY ARRAY)
                  (VARIABLE (LOOP-INDEX)))
       (ARRAY-ELEMENT
```

4

```
(LIST OF NUMBERS)
(POINTER-EXPRESSION
 ((ODD-POSITIONS) (LIST OF NUMBERS)
                  (VARIABLE (LOOP-INDEX)))))))
```

This goal invokes the production, GOAL-SET-EQUAL-1, which calls CODE-GEN with the template, EQUAL-1, to produce the code for setting the current element of the auxillary array equal to the next odd number in the list of numbers.

Creating the code for this requires setting two subscripted variables equal to each other. Normally, subscribted variables are handled identically to simple variables: the decription of the variable is passed to CODE-EL by CODE-GEN. If both the variable and subscript have been used before and are in MEANINGS then CODE-EL returns as a "name" the complete subscripted variable name, including parenthesis and subscript variable. Thus, if CODE-EL is given:

```
(ARRAY-ELEMENT (LIST OF NUMBERS) (VARIABLE (LOOP-INDEX)))
```

it returns

```
L(I)
```

If either the array name or the subscript name are not in MEANINGS, then CODE-EL reports failure and returns NIL for the whole expression. This mechanism is intended to represent a psychological reality, that if the subscript is known, the subject treats subscripted variables identically to simple ones.

In this case, the expression for the current element of the auxillary array is:

```
M(I)
```

Since both the array name and the subscript are known, this is generated in the manner just described. The expression for the next odd element in the list of numbers represents much more of a problem; not only is the subscript not in MEANINGS, but it is a complex expression rather than just a variable name. The model handles this by creating a goal for the whole subscripted variable name when CODE-EL fails:

```
(GOAL
  (ARRAY-ELEMENT
   (LIST OF NUMBERS)
   (POINTER-EXPRESSION
    ((ODD-POSITIONS) (LIST OF NUMBERS)
                (VARIABLE (LOOP-INDEX))))))
```

This goal is satisfied by GOAL-ARRAY-ELEMENT-1 which calls CODE-GEN with template, ARRAY-ELEMENT-1. The process of creating code for the template in turn produces:

```
(GOAL
  (POINTER-EXPRESSION
   ((ODD-POSITIONS) (LIST OF NUMBERS)
              (VARIABLE (LOOP-INDEX)))))
```

This goal is satisfied by the GOAL-POINTER-EXPRESSION-ODD-POSITIONS-1 production which then permits completion of coding of the ARRAY-ELEMENT-1 template. In turn, this allows completion of the EQUAL-1 template to produce:

```
M(I) = L (2 * I + 1)
```

An analogous process is visible in the protocol. In lines 283-284, he writes the first part of the statement up to the subscript for L:

```
S283:SAY M SUB I IS GOING TO BE EQUAL TO L SUB
A284:[WRITES M(I) = L( ]
```

Lines 285-292 are concerned with the creation of the subscribt and correspond to the goal for a pointer expression in the program:

```
S285:ALL RIGHT, NOW WE NEED A POINTER GOING  BACKWARDS
S286:BUT THAT DOESN'T MATTER SO MUCH
S287:YEAH, HERE'S SUB I
S288:AHH, IS THERE A TRIVIAL INVERSE
S289:GO FROM ONE TO 50
```

S290:THAT'S JUST GONNA BE THE ODD NUMBERS
S291:SO THAT'S L 2 TIMES I PLUS ONE
A292:[COMPLETES STATEMENT WITH (2 * I + 1)]

The effect of this piece of code, again, combined with the plan element, are the conditions for the production, COPY-LOOP-TERMINATION-1, which produces

```
(GOAL
  (LOOP-END
   (LABEL
    (LOOP-THROUGH
     ((LIST OF NUMBERS)
      ((COPY ((ODD-POSITIONS) (LIST OF NUMBERS))
            ((FIRST-HALF) (AUXILLARY ARRAY)))))))))
```

The production, GOAL-LOOP-END-1, responds to this goal by calling CODE-GEN on the template, LOOP-END-1, to generate the CONTINUE statement which closes the loop. This completes the plan element.

The code which has been completed up to this point is:

```
DO 10 I = 1 , 50
M ( I ) = L ( 2 * I + 1 )
10 CONTINUE
```

The creation of this piece of code by the subject takes place in a slightly different order than in the program. The subject writes the beginning of the DO statement, but decides to figure out later what the upper bound of the loop should be:

S282:LET'S WORRY ABOUT WHERE IT'S GONNA GO TO LATER

He then begins to write the statement for setting the two array elements equal and interrupts what is doing to go back and complete the DO statement:

S289:GO FROM ONE TO 50
S290:THAT'S JUST GONNA BE THE ODD NUMBERS

Since the subscript expression depends on the loop bound, this suggests that the subject has deferred deciding on the loop bound until after he has decided on the subscript expression. The model lacks mechanisms for deferring decisions in this manner so that it works in the reverse order, using the loop bound to determine the subscript.

The next plan element is:

```
(PLAN-ELEMENT  (COPY ((EVEN-POSITIONS) (LIST OF NUMBERS))
                     ((SECOND-HALF) (AUXILLARY ARRAY)))
```

This plan element is very similar to the previous one, and generation of code for it takes place in a very similar manner. Again, however, there is a discrepancy between the way in which the program generates code and the way in which the subject does. For this loop, the subject first tries making the index increase by steps of 2 with an upper bound of 100 and using 50 + I/2 as the subscript expression. He eventually decides to do it with a loop running from 1 to 50 and using I as the subscript. Functionally, this is completely equivalent to the way the model finally writes the program.The subject's rationale for trying this alternative is that it may be more efficient in terms of machine time:

S310:DIVISION IS A LITTLE BIT SLOWER THAN MULTIPLICATION
S311:WE'RE STILL DOING THE SAME NUMBER OF STEPS
S312:SO THAT'S NO GOOD
S313:THAT DOESN'T SAVE ANYTHING

Efficiency issues such as this are presumably handled as a special kind of symbolic execution in which the emphasis is on certan side effects of the code structure - memory utilization, etc. The program does not incorporate this specialized symbolic execution and so does not attempt this altenative coding.

The succeeding plan element is:

```
(PLAN-ELEMENT (COPY-AND-SET
  (((FIRST-HALF) (AUXILLARY ARRAY))
   ((FIRST-HALF) (LIST OF NUMBERS)))
  ((AUXILLARY ARRAY) (EVEN-POSITIONS))))
```

It calls for copying the first half of the auxillary array into the first half of the array for the list of numbers and setting the elements in the first half of the auxillary array to the numbers of the even positions in the original array. This is first responded to by the production, COPY-AND-SET-LOOP-THROUGH-1, which produces the goal,

```
(GOAL
  (LOOP-THROUGH
   ((AUXILLARY ARRAY)
    ((COPY-AND-SET
      (((FIRST-HALF) (AUXILLARY ARRAY))
       ((FIRST-HALF) (LIST OF NUMBERS)))
      ((AUXILLARY ARRAY) (EVEN-POSITIONS))))
   NIL
   (VALUE ((FIRST-HALF) (AUXILLARY ARRAY)))))
```

This is responded to by the production, GOAL-LOOP-THROUGH-2. This production calls CODE-GEN to create a DO-loop statement.

The effect of this loop in the context of the plan element invokes COPY-AND-SET-EQUAL-ARRAY-HALVES-1; the goal created by this production is satisfied by GOAL-SET-EQUAL-1 which creates the goal for setting the element of the array for the list of numbers equal to the corresponding element in the auxillary array.

The next step in carrying out the plan is:

```
(GOAL
  (SET (ARRAY-ELEMENT (AUXILLARY ARRAY)
                 (VARIABLE (LOOP-INDEX)))
     (POINTER-EXPRESSION
      ((EVEN-POSITIONS) (AUXILLARY ARRAY)
                 (VARIABLE (LOOP-INDEX))))))
```

It is a goal for setting the elements in the auxillary array equal to the numbers of the odd positions in the list. It is responded to by GOAL-SET-EQUAL-1 which uses CODE-GEN to attempt to create the necessary code. In the process of creating this code the goal:

```
    (GOAL
       (POINTER-EXPRESSION
        ((EVEN-POSITIONS) (AUXILLARY ARRAY)
                       (VARIABLE (LOOP-INDEX)))))
```

must be satisfied; it is a goal for an expression which is the value of the even positions in the arrays. When this goal is satisfied and the rest of the template for setting the two things equal can be completed the code produced is:

```
    M(I) = 2 * I + 1
```

Completion of this code is that last action that must be performed inside the loop. A goal for loop-termination creates the CONTINUE statement and completes action on the plan element.

The corresponding segment of protocol is:

S325:DO 30
S326:I GOES FROM ONE TO 100
A327:[WRITES DO 30 I=1,100]
S328:L SUB I EQUALS M SUB I
A329:[WRITES L(I) = M(I)]
S330:AND THAT M SUB I IS NOW EQUAL TO THE SAME THINGS AGAIN
S331:NOW WE HAVE THE INVERSE FUNCTION
S332:IF IT'S EVEN THEN IT'S TWO TIMES I PLUS ONE
S333:IF IT'S ODD THEN IT'S THAT
S334:LET'S DO IT ONE TO 50
A335:[CHANGES RANGE OF DO LOOP TO 1,50 FROM 1,100]
S336:M SUB I IS GOING TO BE 2 TIMES I PLUS ONE
A337:[WRITES M(I) = 2 * I + 1]

In comparing the model to it, note that in the model, a separate goal is created for the expression that is to be the new value for M(I). An equivalent behavior is seen in lines 330-333. However, the subject's behavior has a consequence, changing the value of the upper bound of the array, which the model does not produce. This behavior on the subject's part could have come about in either of two ways: One is as a varient of the deferred decision behavior seen in the previous plan element; the value of 100 could be just a temporary place holder until the real value is worked out. Alternatively, he could have intended to write 50 all along, a view somewhat supported by his use of I, rather than (I+1)/2, as the subscript in the previous line. If the former is the case, then the model is unable to produce the behavior. If the latter is true, though, the LOOK-AT-CODE mechanism, together with appropriate retrievals from MEANINGS, could be used to obtain the same effect.

The final element of the plan is:

```
(PLAN-ELEMENT  (COPY-AND-SET
 (((SECOND-HALF) (AUXILLARY ARRAY))
  ((SECOND-HALF) (LIST OF NUMBERS)))
 ((AUXILLARY ARRAY) (ODD-POSITIONS)))))
```

Coding of it begins in a very similar manner to the previous plan element.

The code created by the program up to this point is:

```
DO 10 I = 1 , 50
M ( I ) = L ( 2 * I + 1 )
10 CONTINUE
DO 20 I = 1 , 50
M ( I ) = L ( 2 * I )
20 CONTINUE
DO 30 I = 1 , 50
L ( I ) = M ( I )
M ( I ) = 2 * I
30 CONTINUE
DO 40 I = 1 , 50
L ( I + 50 ) = M ( I + 50 )
M ( I ) = 2 * I + 1
40 CONTINUE
```

This code is identical to the code generated by the subject up to this point, including the error in the subscript for M in the second-to-last line.

As in the previous two examples, the size of units in which the model generates code corresponds fairly well to the size of unit seen in the protocol. In this case, however, the model has more difficulty with the order in which code is generated; it is unable to duplicate an attempt to find a more efficient way of coding an operation, and it has no mechanism for deferring decisions about loop bounds. Cure of the first problem would depend on a better representation of written code than the current CODE structure, since an recoding for efficiency would have to be driven by the code already written. Cure for the deferred decisions problem is probably simpler and would involve the addition of productions to handle the case in which a goal is not immediately satisfied.

4.4 Problem LARRY

Problem LARRY was the eighteenth problem completed by the subject; it had the shortest solution time of any problem in the set, taking only 4.7 minutes to write and 4.9 to de-bug, and was, therefore, selected as one of the problems to model. The task for the subject to program was to find all the items in the list that ended in one in base 10 notation and place them at the beginning of the array; the postions in M corresponding to these numbers were to be set to 1. The complete protocol is given in Appendix 2.

The problem is a good example of a straight procession through the three processes with no back-tracking to a previous process. The understanding process consists simply of reading the instructions, in the first 7 lines of the protocol. The planning process is slightly more complex. It begins with planning a solution to a sub-problem, how to test whether a number ends in the digit, one (lines 8-18). The solution that is found is:

1. Divide the number by 10 and then multiply it by 10.
2. Subtract this quantity from the original number.
3. The difference will be the digit that the number ends in.

With this subproblem solution in hand, the subject creates the plan for solving the main problem. The initiation of this process can be seen in lines 19-24 of the protocol. Since there are no other planning segments in the protocol, the plan created at this point is, presumably, the one used for coding.

Using the statements in these lines and the code that he actually generates, it can be inferred that his entire plan might appear as follows:

1. Create a pointer to keep track of the number of numbers found which meet the condition.

2. Go through the array and find all numbers which meet the condition. When one is found,

   Increment the pointer for number of numbers found.

   Swap the element the pointer now points to with the number just found.

In the program, this plan is represented as:

```
(CREATE (POINTER ((LAST-MODULO-FOUND) (LIST OF NUMBERS))) 0)
 (FIND-ALL ((1 MODULO 10) (LIST OF NUMBERS)))
 (BEGIN! (1 -MODULO-10-FOUND))
 (INCREMENT (POINTER ((LAST-MODULO-FOUND) (LIST OF NUMBERS))))
 (SWAP (ARRAY-ELEMENT
        (LIST OF NUMBERS)
        (POINTER ((LAST-MODULO-FOUND) (LIST OF NUMBERS))))
       (ARRAY-ELEMENT (LIST OF NUMBERS) (VARIABLE (LOOP-INDEX))))
 (SET (CORRESPONDING-ELEMENT
       (AUXILLARY ARRAY)
       (VARIABLE (LOOP-INDEX)))
      1)
 (END! (1 MODULO- 10 -FOUND))
 (END! (FIND-ALL-LOOP-THROUGH))
```

4.4.1 Program Operation for Lines 25-59

The first step of this plan is interpreted as "create a pointer to the last number found which had the propei modulo division properties; initialize it to 0." This is responded to by the production, PLAN-CREATE-POINTER-1, which has two effects. The first is to create a new entry in MEANINGS with three attribute-value pairs in it; the NAME, obtained via NEWQUAN, is N, it is of TYPE POINTER, and it MEANS ((LAST-MODULO-FOUND)(LIST OF NUMBERS)). The second is to call CODE-GEN with the template, EQUAL-2, to create the code,

```
N=0
```

The equivalent behavior by the subject is seen in lines 25-29 of the protocol:

S25:DO
S26:ALL RIGHT, SO IT'LL BE NUMBER
S27:EQUALS 0
A28:[WRITES N=0]
S29:THAT'S HOW MANY THERE ARE

The next plan element is:

```
(PLAN-ELEMENT  (FIND-ALL ((1 MODULO 10) (LIST OF NUMBERS))))
```

This requires finding all elements in the list of numbers which leave the remainder 1 when divided by 10. The production which responds to this plan element is PLAN-FIND-ALL-MODULO-2 which creates the goal:

```
(GOAL
   (LOOP-THROUGH ((LIST OF NUMBERS) (FIND-ALL-MODULO-LOOP))))
```

This is a goal for looping through the list of numbers.

The effect of writing the DO loop for this goal, combined with the plan element, invokes FIND-ALL-MODULO-2. This is a production for creating a test for whether a quantity is divisible modulo n when it is desired to find all such items. It creates the goal:

```
(GOAL
(IF (TEST (1 MODULO 10)
       (ARRAY-ELEMENT (LIST OF NUMBERS)
                (VARIABLE (LOOP-INDEX))))
   (GOTO
    (LABEL
     (LOOP-THROUGH
      ((LIST OF NUMBERS)
       (FIND-ALL ((1 MODULO 10) (LIST OF NUMBERS)))))))))
```

This is interpreted as a goal for an IF statement which tests whether a number has a remainder of 1 when divided by 10; if it does, it branches to the label which ends the DO loop. As with the IF statements in the previous examples, the production, IF-TEST-1, is invoked and calls CODE-GEN with the basic IF statement template, IF-1; creation of the particular test within the statment is then handled by a sub-goal created during the course of coding the template. Again, an equivalent behavior appears in the protocol, in lines 34 - 40:

```
S34.MINUS L SUB I DIVIDED BY 10
S35:TIMES 10
S36:SOMETHING  ONE
S37:IF IT'S EQUAL TO ONE
S38:IF IT'S NOT EQUAL TO ONE
S39:WE JUST LOOP
S40:NOT EQUAL TO ONE
```

When the entire template is complete the code that has been created up to this point is:

```
N=0
DO 10 I=1,100
IF(L(I)/10*10 .NE.1) GOTO 10
```

Note that the branch is taken if the number does not have the proper remainder when divided by 10.

The next part of the plan is a block containing the actions to be taken when a number meeting tho criterion is found. In the protocol, it is signaled by the subject's statement in line 43:

S43:OTHERWISE WE'LL SAY

In the program, the block begins with:

```
(PLAN-ELEMENT  (BEGIN! (1 -MODULO-10-FOUND)))
```

which marks the beginning of the block; it is responded to by PLAN-BEGIN-1 which rehearses appropriate items, but does not create any code. The plan element following it is:

```
(PLAN-ELEMENT (INCREMENT (POINTER
          ((LAST-MODULO-FOUND) (LIST OF NUMBERS)))))
```

This results in invoking PLAN-INCREMENT-POINTER-1 which calls CODE-GEN with template, INCREMENT-1, to create the code for incrementing the pointer.

Once this has been completed the next plan element is:

```
(SWAP (ARRAY-ELEMENT
    (LIST OF NUMBERS)
    (POINTER ((LAST-MODULO-FOUND) (LIST OF NUMBERS))))
   (ARRAY-ELEMENT (LIST OF NUMBERS) (VARIABLE (LOOP-INDEX))))
```

This calls for swapping the number which has just been found (and which is pointed to by the loop index) with the number pointed to by the pointer for the last modulo found. PLAN-SWAP-1 is invoked by this plan element, and it calls CODE-GEN with the template, SWAP-1. Note, again, that the swap consists of a single template and is treated as a single FORTRAN statement by the subject:

S46:L L EQUALS L SUB I
A47:[WRITES LL=L(I)]
S48:L SUB I EQUALS L SUB N
A49:[WRITES L(I)=L(N)]
S50:I'M NOT GONNA DESTROY THE ARRAY
S51:L SUB N IS EQUAL TO L SUB I

A52:[L(N)=L(I)]

After the swap is completed, the next plan element is,

```
(SET (CORRESPONDING-ELEMENT
     (AUXILLARY ARRAY)
     (VARIABLE (LOOP-INDEX)))
```

Through the production, PLAN-SET-EQUAL-1 and a call to CODE-GEN with template, EQUAL-1, this produces the code for setting the corresponding element in the auxillary array to 1.

The block in the plan is terminated by

```
(PLAN-ELEMENT  (END!))
```

The production, PLAN-END-1, responds to this. As with the BEGIN, actual code is created from the plan element (though the production does have other actions, such as rehearsing items).

The final element of the plan is

```
(PLAN-ELEMENT  (END! (FIND-ALL-LOOP-THROUGH))))
```

PLAN-END-CODE-CONDITION-2, which responds to this, calls CODE-GEN with the template, LOOP-END-1, to create the CONTINUE statement at the end of the loop. This completes the program. The final code appears as

```
N = 0
DO 10 I = 1 , 100
IF ( L ( I ) / 10 . NE . 1 ) GOTO 10
N = N + 1
LL = L ( N )
L ( N ) = L ( I )
L ( I ) = LL
M ( I ) = 1
10 CONTINUE
```

Again, this code is identical to the program written by the subject.

4.5 Evaluation of the Model for the 4 Segments

In evaluating the adequacy of a model of behavior the answers to two questions must be considered. The first is to what extent does the output of the model corresponds to the data, and, second, what evidence exists for the validity of the mechanisms used by the model to produce the output. The second question is considered at length in the next chapter; the first question is of interest here.

While the main focus of this discussion will be on the comparison at the level of code unit size and sequence of generation, a necessary preliminary inquiry concerns whether plans used in the model accurately represent the plans used by the subject. This question can be broken into two parts: (1) whether the functional form of the model plans is correct; and (2) whether the model plans contain the same level of specification as those used by the subject. An answer to the first part of the question may be obtained through an inquiry into the use of functional language In the protocols. A phrase in the protocols was considered to be a use of functional language if it consisted of a statement of some function or action to be performed. In contrast, statements about about relationships or about states or goals were not classified functional language. If such functional language is a frequent feature of planning segments in the protocols, then it can be argued that the use of similar functional representations in the model accurately reflects the subject's behavior.

In fact, such functional language is a clear feature of the subject's plans. To begin with, the reader may verify its use in the planning segments of the four problems discussed in this chapter. For example, in problem RICHARD, the subject says "**determine** if a number's odd or not" and in JOHN he says, "I'm gonna take every other one and move them over." Across the 41 planning segments in the 23 protocols, functional language can be unequivocally identified in 36 of them, occurring on the average 11 times in every 100 lines of planning. From this use of functional language in the protocols, it is likely that the functional form of plans in the model corresponds to the way in which the subject represents plans.

The answer to the second part is more difficult because, often, a greater level of detail is inferred than is actually directly stated in the protocol. An argument that the greater level of detail seen in the model accurately reflects the level of detail really used by the subject is that verbalizations would be expected to contain less detail than the subject is in fact using. This argument is, however, only a tentative one and further research is necessary to confirm it.

Having discussed the issue of plan correspondence, the remaining question is that of how veridically the model does code generation. A desirable type of answer would be some sort of overall index measuring the correspondence. Unfortunately, constructing such an index is very difficult in this case. Perhaps because the task is a design problem, there is no readily available way to reduce the protocols to sequences of smaller episodes for scoring purposes. Construction of some global order measure based, say, on information theoretic grounds also appears out of the question. Instead, evaluation of the model-subject correspondence will have to be made on other grounds.

The first of these grounds is simply the observation that the model has substantial problems on only one, problem JOHN, out of the four protocols. Even the difficulties with this problem can be characterized more as a failure to explore some alternative paths rather than as producing radically differeing code. In the other three problems, there are few or no major mis-matches.

A second ground is in the ratio between the number of lines in the protocol and the number of cycles taken by the production system to model that behavior. These ratios, which provide a crude time or effort comparison, are shown in the following table:

Name	Lines	Cycles	Ratio	Code
LARRY	38	60	1.6	9
RICHARD	47	65	1.4	10
JOHN	80	169	2.1	11
LEE	89	110	1.2	13

Table .2

Problem names are given under the Names heading. The entries under the Cycles heading are the total number of cycles of the production system necessary to create the code for the segment; those under the Lines heading are the number of lines of protocol included in the segment while those under the Code heading are the number of lines of code generated in the segment by both the subject and the program. The ratio column gives the ratio between the number of cycles of the production system and the number of lines of protocol in the protocol segment. (No ratios for lines of code versus production system cycles are given since the number of lines of code generated is not a good indicator of "work done" by either the subject or the system.) Note that 3 of 4 of the ratios lie very close together. Only problem JOHN appears out of line, probably because of the difficulties the model has in matching the behavior of the subject. This rough equivalence of the ratios is a further indication of the correspondence between the operation of the model and the behavior seen in the protocols.

4.6 A Note on STM Size

During the discussion of the STM in the description of the model, it was stated that the size was set at 14 on a largely arbitrary basis since neither current literature nor evidence from the protocols presented a basis for a different figure. To investigate whether 14 was a wise choice the program was run with various sizes of STM for problem JOHN, the problem with the most elaborate sub-goal structures. When an STM of 7 elements was used, the current plan element was lost off of the end of STM by the end of only the 11th cycle of the production system. With an STM of 9 elements, both the current plan element and the goal created from that plan element were lost by the 29th cycle. Using an STM of 11 elements, the production system ran for 38 cycles before losing a goal that still had not been satisfied. With 12 elements, the same problem occurred as with 11 elements. 13 elements allowed the model to run satisfactorily.

To a certain extent, the necessary size of STM is a function of rehearsal strategies; if these strategies allow much outdated information to remain near the front of STM, then a large STM will be necessary to prevent needed information from being pushed off the the end. To check whether this was a problem in this case, the maximum number of "active" elements that had to b in STM for the program to run was determined by observing which elements were necessary for subsequent productions. At the 28th cycle of the production system, 10 elements fell into this classification. Thus, even with optimum rehearsal strategies, STM had to be at least 10 elements long.

From these calculations, 14 appears to be slightly too large for the size of STM. In general, the system indicates that it is done with information by marking it in some way, for example, by changing NEW-CODE to OLD-CODE, rather than getting rid of it by pushing off of the end of STM. The only effect of the retention of additional information that the larger STM size permits is, therfore, restricted to allowing less than perfect rehearsal strategies.

5

Analysis of the Model

In the preceeding chapter, a correspondence between the output of the mofel and the behavior in the protocols was demonstrated. For the model to be considered valid, a remaining, necessary condition is that evidence be presented for the plausibility of the mechanisms which produce the model's behavior. The basis for this requirement is that these mechanisms are, in fact, the locus of the psychological assertions which the model makes. Like any other psychological assertions, they must be judged not only on the basis of how well they fit particular observed data but also how consistant they are with other related data and findings. The model here makes use of three primary mechanisms to which correspond equivalent assertions:

1. The production system consists to a substantial extent of productions which have particular plan elements as their invoking conditions and whose actions are instructions on how to create code for those particular plan elements. This asserts that knowledge about programming consists of a large amount of specific information about how to carry out in programming languages various operations not implemented as language primitives.

2. Code creation in the model is accomplished by the production system, CODE-GEN function, and MEANINGS structure operating in a manner described as symbolic execution. The equivalent psychological assertion is that code creation by the subject also is accomplished by symbolic execution.

3. Knowledge about a programming language's syntax is encoded in the model in the form of coding templates ranging in size from a sub-expression up to a few lines in length. Human programmers are asserted to use similar sized units.

The following sections provide evidence for the plausibility of these assetions.

5.1 Support for the Assertions about Knowledge Representation

5.1.1 Analysis of Frequency of Production Use

In the program the linkage between a particular part of the plan and the code that is eventually generated from it is effected entirely through the production system. The following analysis of the frequency of production use displays the nature of this linkage:

Summing across the four problems, the production system ran for a total of 404 cycles; 73 productions were each used at least once. (i) Usage statistics for these 73 productions are shown in the following table:

Production Usage

Number of Uses	Number of Productions	Number of Cycles	% of Total
1-2	47	64	15.8
3-8	9	43	10.6
>9	17	297	73.5

Table 5.1.1.1

Five productions, CONTINUE-CODE-GENERATION-1, EQUIVALENT-EFFECT-1, LOOK-AT-CODE-1, NEW-CODE-2 and NEW-CODE-3, accounted for 216 cycles, or 53.5% of the total. Viewed across the four segments, 21 of the 73 productions (28.8%) were used in

----

(i) The production system given in Appendix 5 actually consists of 81 productions; 8 of these were included on an a priori basis but were not actually used in the four problems chosen. They were EQUIVALENT-EFFECT-2, GOAL-LENGTH-1, GOAL-POINTER-1, GOAL-TEST-ODD-PARITY-1, GOAL-TEST-POSITIVE-1, INSERT-BEFORE-1,NEW-CODE-4, and PLAN-BEGIN-CODE-CONDITION-1.

2 or more of the segments. They account tor 81.7% of the total cycles. Summarizing these figures, they indicate production use in this model has a "Zipf's law" distribution, in which the nth most frequently used production is used c/n times as often as the first one (Knuth, 1973). In other words, the system contains a large number of special purpose productions which are used very infrequently.

On the basis of content, the productions can be divided into 4 groups. The first group are productions for overall control and goal management and include the LOOK-AT-CODEs, the EQUIVALENT-EFFECTs CONTINUE-CODE-GENERATION-1, MAKE-OLD-PLAN-ELEMENT-1, and NEXT-PLAN-ELEMENT-1. The second group contains all the productions which add to or modify code, including all the NEW-CODEs, ADD-ON-CODE-1, and REPLACE-CODE-1. The third group includes all the GOAL productions, e.g., GOAL-TEST-POSITIVE-1. The fourth group contains all the PLAN productions, e.g., PLAN-CREATE-POINTER-1. All the productions in the first group (except EQUIVALENT-EFFECT-2) are used in all 4 segments. Of the second group, 3 of the 8 are used in more than one problem. In the third group, 6 out of 29 productions are used in multiple segments. In the last group, however, only 3 out of 35 productions are used in several segments. Also, the five most commonly used productions given earlier belong only to the first two groups. What can be concluded from these statements is that the special productions are not evenly distributed but are, instead, primarily plan productions and, to a lessor extent, goal productions.

As mentioned in the introduction to this section, this distribution of productions is equivalent to a psychological assertion: An experienced programmer knows a large number of distinct plan elements and has associated with each one, information on how to code it. Each production or sequence of productions, if several are required to code a plan element, is a representation one of these pieces of knowledge, and the large number of these productions is a reflection of the size of the body of knowledge they represent.

5.1.2 Evidence from Plans in Other Problems

Important evidence on the truth of this proposition would be the total set of plan and goal productions necessary to simulate behavior for all the coding segments in all the 23 protocols. If a substantial number of new plan and goal productions were necessary to extend the existing production system to model them and if it were difficult to collapse these new productions into fewer, more general ones, then this would support the assertion about the necessity of a large number of separate knowledge elements.

Even though the entire set is not available, an estimate of the number of productions and some idea of their type is obtainable from just the plans for some of the protocols. The procedure that is followed is to find for each plan element whether a production already exists or whether an existing production can be easily modified to fit; if none match, then a new production or sequence of productions is required. Though it gives no indication of what productions will be required for sequencing among the plan elements, it does give an idea of what basic productions will be necessary.

For this purpose, four problems, WILLIAM, CARL, ROBERT, and FRANK, were selected at random from those problems set aside before creation of the model was begun. (Complete listings of these protocols are given in Appendix 2.) Their plans were specified from the protocols and from the code that was actually created in the same manner as was done with the 4 segments in the previous chapter; if more than one plan was used, the plan that resulted in the running program was selected. Each of the following sections discusses the plan for each of these problems and the productions that would probably be necessary to carry it out.

5.1.3 Plan for Problem WILLIAM

The initial plan for problem WILLIAM is:

```
1.  (FIND-ALL ((SEQUENCES (INCREASING)(LIST OF NUMBERS)))
2.  (BEGIN! (SEQUENCE-FOUND))
3.  (IF (TEST ((GREATER) (LENGTH (LONGEST-SEQUENCE-FOUND))))
                        (LENGTH (CURRENT-SEQUENCE))))
      (GOTO (LABEL (FIND-NEXT-SEQUENCE)))
4.  (BEGIN (OTHERWISE!))
5.  (SET (VARIABLE (BEGINNING (LONGEST-SEQUENCE-FOUND)))
       (VARIABLE (BEGINNING (CURRENT-SEQUENCE))))
6.  (SET (VARIABLE (END (LONGEST-SEQUENCE-FOUND)))
       (VARIABLE (END (CURRENT-SEQUENCE)))
7.  (SET (VARIABLE (LENGTH (LONGEST-SEQUENCE-FOUND)))
       (VARIABLE (LENGTH (CURRENT-SEQUENCE))))
8.  (END! (OTHERWISE!))
9.  (END! (SEQUENCE-FOUND))
10.  (END! (FIND-ALL-LOOP-THROUGH))
11. (LOOP-THROUGH (AUXILLARY-ARRAY)
                (VARIABLE (BEGINNING (LONGEST-SEQUENCE-FOUND)))
                (VARIABLE (END (LONGEST-SEQUENCE-FOUND))))
12. (SET (CORRESPONDING-ELEMENT (AUXILLARY ARRAY)
                          (VARIABLE (LOOP-INDEX)))
       1)
13.  (END! (AUXILLARY-ARRAY-LOOP-THROUGH))
```

The first part of this plan is expanded into:

```
1a.  (LOCATION (FIND-NEXT-SEQUENCE))
1b.  (FIND-EXISTENCE ((SEQUENCE)
                    ((LIST OF NUMBERS)
                     (VARIABLE (CURRENT-LOW))
                     (LENGTH (LIST OF NUMBERS)))))
1c.  (IF (TEST ((GREATER)
               (ARRAY-ELEMENT (LIST OF NUMBERS)
                             (PLUS (VARIABLE (LOOP-INDEX)) 1))))
               (ARRAY-ELEMENT (LIST OF NUMBERS)
                             (VARIABLE (LOOP-INDEX)))
       (GOTO (LABEL (SEQUENCE-FOUND)))
1d.  (END! (FIND-EXISTENCE-LOOP-THROUGH))
1e.  (BEGIN! (NEXT-ELEMENT-FOUND))
1f.  (SET (VARIABLE (LENGTH (LONGEST-SEQUENCE-FOUND)))
        (SUM (PLUS (VARIABLE (LOOP-INDEX)) 1)
                  (VARIABLE (CURRENT-LOW))))
```

Plan-element 1a is used to indicate the subject's intention that the location for finding the next sequence is to be located at this point; a new production would be needed to carry it out. Element 1b could be handled by allowing the invoking conditions for PLAN-FIND-EXISTENCE-FIRST-POSITIVE-1 to be either (FIRST POSITIVE) or (SEQUENCE). Elements 1c to 1e. can be handled by existing productions. Element 1f will, however, require a new GOAL production for the SUM operation. Though the addition operator in FORTRAN is used very frequenctly, it is almost always in the context of an INCREMENT operation or as part of an expression that is used as a simple variable so that no general SUM production was included in the basic production system. Of plan elements 2 to 13, the only new production that is needed is for the LENGTH function in the 3rd element. (In fact, such a production was provided, but not used, in the original production system.) At least 3 more productions are, therefore, necessary to handle this plan.

5.1.4 Plan for Problem CARL

The plan for problem CARL that is used at the beginning of code generation is:

```
1. (CREATE (VARIABLE (MULTIPLICAND-SUBSCRIPT)) 1)
2. (RUN-THROUGH  (VARIABLE (MULTIPLIER))
              NIL
              (NUMBER-OF (MULTIPLIERS) (LIST OF NUMBERS)))
3. (SET (VARIABLE (MULTIPLIER-LIMIT))
       (NEXT-ONE-AFTER (SUM (VARIABLE (MULTIPLICAND-SUBSCRIPT))
                            (VARIABLE (MULTIPLIER)))))
4. (LOOP-THROUGH (LIST OF NUMBERS)
              (VARIABLE (MULTIPLICAND-SUBSCRIPT))
              (VARIABLE (MULTIPLIER)))
5. (SET (ARRAY-ELEMENT (LIST OF NUMBERS)(VARIABLE (LOOP-INDEX)))
       (PRODUCT
        (ARRAY-ELEMENT (LIST OF NUMBERS)(VARIABLE (LOOP-INDEX)))
        (VARIABLE (MULTIPLIER))))
6. (SET (CORRESPONDING-ELEMENT (AUXILLARY-ARRAY)
                             (VARIABLE (LOOP-INDEX)))
```

```
      (VARIABLE (MULTIPLIER)))
7. (END! (LOOP-THROUGH (LIST OF NUMBERS)))
8. (INCREMENT (VARIABLE (MULTIPLICAND-SUBSCRIPT))
          (VARIABLE (MULTIPLIER)))
9. (END! (LOOP (VARIABLE (MULTIPLIER)))
```

The RUN-THROUGH construction in element 2, like the LOOP-THROUGH, produces a DO loop construction when coded; it differs from it in the programmer's intent. In this construction, his primary intent is to go through a set of possible variable values; in the LOOP-THROUGH, the primary intent is to act on each of the elements of an array or vector, and the change in values of the loop index is only incidental. (There are programming languages in which a single statement is used to cause a function to be applied to all the elements of an array; in these languages, the RUN-THROUGH would continue to exist while the LOOP-THROUGH operation would be unnecessary.) Since the RUN-THROUGH construction is not used in the first 4 segments, a new production would be required for it.

The NUMBER-OF construction, used to indicate the need for an actual numeric value, also requires a production. Another production is needed for the NEXT-ONE-AFTER construction, used to indicate the need for an expression which points to the next element after the current one in an array. Finally, a production is needed for the PRODUCT function in element 5; as was the case with the SUM function, this operation was not included in the original set because it was only used inside another expression and never as an isolated operation. Altogether, at least 4 new productions are required for this plan.

5.1.5 Plan for Problem ROBERT

The basic plan in this problem is:

```
1. (ORDER-POINTERS (LIST OF NUMBERS))
2. (CREATE (VARIABLE (NUMBER-OF-SEQUENCES))) 1)
3. (INITIALIZE (CORRESPONDING-ELEMENT
```

5

```
                  (AUXILLARY ARRAY)
                  (ARRAY-ELEMENT (POINTER ARRAY) 1)
                  1)
4.  (SET (POINTER (LOW-POINT))
          (ARRAY-ELEMENT (LIST OF NUMBERS)
                         (ARRAY-ELEMENT (POINTER ARRAY) 1)))
5.  (LOOP-THROUGH (POINTER ARRAY)
                  (PLUS (BEGINNING (POINTER ARRAY)) 1))
6.  (IF (TEST (LESS) (ABSOLUTE-VALUE
                      (DIFFERENCE
                       (ARRAY-ELEMENT
                        (LIST OF NUMBERS)
                        (ARRAY-ELEMENT
                         (POINTER ARRAY)
                         1))
                       (POINTER (LOW-POINT)))
                      11)
         (GOTO (LABEL (CONTINUE-SEQUENCE)))
7.  (BEGIN! (OTHERWISE))
8.  (SET (POINTER (LOW-POINT))
          (ARRAY-ELEMENT (LIST OF NUMBERS)
                         (ARRAY-ELEMENT
                          (POINTER ARRAY)
                          (VARIABLE (LOOP-INDEX)))))
9.  (INCREMENT (VARIABLE (NUMBER-OF-SEQUENCES)))
10.  (END! (OTHERWISE))
11.  (SET (ARRAY-ELEMENT
            (AUXILLARY ARRAY)
            (ARRAY-ELEMENT (POINTER ARRAY)
                           (VARIABLE (LOOP-INDEX))))
           (POINTER (LOW-POINT)))
12.   (END! (LOOP-THROUGH (POINTER-ARRAY)))
```

The first element of this plan requires ordering a set of pointers to the list of numbers by the values of the numbers in the list. Since the subject uses a different sorting algorithm to sort the pointers than he uses to sort array elements directly, the PLAN-ORDER-1 production cannot be used, even if it is modified to change pointer values. At least 7 new productions are necessary to handle the sequence of steps for this pointer sort. Additionally, a GOAL production to create an expression in which an array element is used as a subscript to another array element is probably also necessary, so that just the first element of this plan uses 8 new productions.

Additionally, plan-element 6 requires 3 more productions, one each for the test for less, the absolute value functon, and the difference operation. Altogether, this plan requires at least 11 more productions.

5.1.6 Plan for Problem FRANK

The plan is:

```
1.  (FIND-SUM (LIST OF NUMBERS))
2.  (SET (VARIABLE (AVERAGE))
        (DIVIDEND (VARIABLE (SUM))(LENGTH (LIST OF NUMBERS))))

3.  (LOOP-THROUGH (LIST OF NUMBERS))
4.  (IF (TEST ((LESS-EQUAL)(ARRAY-ELEMENT (LIST OF NUMBERS)
                                  (VARIABLE (LOOP-INDEX))))
       (GOTO (LABEL (LOOP-END)))
5.  (BEGIN! (OTHERWISE))
6.  (SET (CORRESPONDING-ELEMENT (LIST OF NUMBERS)
                         (VARIABLE (LOOP-INDEX))
        1)
7.  (SET (ARRAY-ELEMENT (LIST OF NUMBERS)(VARIABLE (LOOP-INDEX))
        (PRODUCT (ARRAY-ELEMENT (LIST OF NUMBERS)
                          (VARIABLE (LOOP-INDEX)))
               (ARRAY-ELEMENT (LIST OF NUMBERS)
                          (VARIABLE (LOOP-INDEX))))
8. (END! (LOOP-THROUGH (LIST OF NUMBERS)))
```

Item 1 requires the finding of the sum of the list of numbers; it is represented as a single plan element because the subject treats it as a single operation. Since it requires the creation of two separate pieces of code, at least two productions would be necessary to carry it out. The DIVIDEND operation in element 2 and the less-equal test in element 4 each also require a production. Overall, 4 more productions are necessary for this plan.

5.1.7 Conclusions from the 4 Plans

These four new plans require increments of 3, 4, 11, and 4 new productions respectively. This figure is based on an assumption of complete overlap, with each new plan having available all the productions used in all prior plans. Again, it should be emphasized that this estimate is a low one because it omits productions which may be necessary to sequence between the the productions for the individual plan elements.

Another point to also note is that a priori the number of new productions required for each new segment ought to decrease, especially after the first few, as the system acquires more knowledge that can be used across situations. Even ignoring these factors, however, the number and type of new productions needed for these new plan elements is an impressive indication of the large amount of specific knowledge about plans which this programmer has and which must be included in the production system. (It should also be emphasized that this finding is not just an artifact of a lack of generality in the particular production system used. Even if a production system were used which had more general productions, say, as the by-product of a more sophisticated use of variables in the matching process, the same specific information about how to code specific plans would still have to be built into the system.)

An appropriate perspective in which to view this finding can be obtained by comparing the number of productions required for this task (73) with that required for two other tasks, cryptarithmetic (Newell & Simon, 1972) and a visual imagery task (Moran, 1973). The cryptarithmetic task required 14 basic productions, (though these productions called on 96 different actions). The visual imagery task used 113 productions, exclusive of those involved in producing verbalizations.

Not surprisingly considering task complexity, the model presented here required several times as many productions as were required for the cryptarithmetic task. Additionally, while essentially the same set of productions would probably work for another cryptarithmetic problem by the same subject, the programming model requires more productions for each new problem.

This latter difference is consistant with the distinction between problem-space problems, such as cryptarithmetic, and design tasks such as programming. The production system used in the visual imagery task provides an additional datum on the number of productions for design tasks. To begin with, note that it is of roughly the same size as the production system used in this model. The important question remaining is whether this same, fixed set could also be applied to other visual imagery tasks. While the author argues that his system is a general one which would require relatively few modifications to handle other tasks, he does leave the possibility open that many more productions would be necessary to handle certain types of information; for example, he asserts that his RECOGNIZE rules are "just excerpts from a large, but straight-forward knowledge-base of facts about spatial configurations" (Moran, 1973, p.150). This suggests that a very large number of productions, each representing a specific fact, as was the case in this programming model, is typical of the whole class of desgin tasks.

5.2 Support for Assertions about Symbolic Execution

Symbolic execution was defined as a method of coding in which as each line of code is generated an effect is assigned to it. The effect is stated in terms of the distinctions that the program must make to acheive its purpose, so that the entire process is equivalent to executing the code for symbolic data representing these distinctions.

The sufficiency of this assertion has already been demonstrated by the ability of the program, which uses this mechanism, to generate code matching that generated by the subject. Verification of the necessity of this mechanism for explaining human code generation is obtained by showing that effects of the type described above are used continually throughout the protocols. For the protocols presented in Appendix 2, the reader is invited to satisfy himself that this is the case. To provide additional support for the assertion that this mechanism is a universal one, an experienced FORTRAN programmer was asked to judge whether symbolic execution took place in some of the remaining protocols not included in the appendices. He was trained by first having him read a definition of symbolic execution and acquaint himseif with it. In addition, he was given the following guidelines:

A statement in the protocois was to be considered as showing symbolic execution if

1. The statement contains phrases such as "this does this," "at this point we know .....," or "if this is so then do this" which express either an action that has been performed or a state that has been reached.

2. It was a statement of some effect that had already been accomplished and was not just a statement of an intent to perform some action.

3. It was not just reading aloud the code that he had written.

Following training on portions of the first 5 protocols given in Appendix 2, the judge was asked to go through each of the sections of coding behavior in each of the other protocols and find all lines of the protocol which met the guidelines given above. He was able to find clear evidence of symbolic execution in 11 out of the 18 coding segments that he look at. The task was also performed by the experimenter who found it in 29 of the 42 coding segments in the remaining protocols. Two things should be kept in mind in Interpreting these results. First, the criteria used were very stringent ones; only verbalizations which could not be interpreted in any other way were scored. Second, some of the segments were very short (as little as 5 lines) and did not include much opportunity for verbalization of symbolic execution, if it occurred. Considering these factors and noteing the frequency of symbolic execution which was observed, it is likely that symbolic execution is a ubiqitious feature of this set of protocols.

5.3 Support for Assertions about Syntax Knowledge

In the presentation of the model, it was asserted that a programmer's knowledge of the syntax of a particular programming language is represented as a collection of small pieces of code, each of which accomplishes some particular operation in the language; the term that has been used to refer to them was "templates." The total number of templates required to represent an actual programmer's knowledge of a given language is important evidence for the validity of the template concept. If the number is very large, on the order of several thousand, then acquiring just the ability to write syntactically correct statements in a new programming language ought to be a task of great difficulty. While no experimental studies have been done on this point, anecdotal evidence suggests that the syntax is, in fact, the easiest part to acquire. This implies that if thousands of templates are required to represent syntactical knowledge, that templates are probably not a good structure for representing this knowledge.

On the other hand, if only a few templates, say, less than a dozen, are adequate for representing a programmer's syntactic knowledge, then the concept also is in difficulty. considering the large number of possible lines or pieces of code that can be constructed then a small number of templates would indicate that what really exists is something like a syntax generator which uses the templates as general syntax specifiers and creates from them an exact syntax specification for the desired statement. Since the protocols contain no evidence whatsoever that anything like a syntax-only generator exists, the occurence of only a small number of templates would indicate that templates as defined in this model are not a good representation of a programmer's syntactic knowledge.

Using the model for the 4 segments and the other protocols an order of magnitude estimate may be obtained for the number of templates required to represent the subject's total knoledge. A total of 18 templates are require for the 4 segments, but several syntactic structures are not included in the segments which do appear in other protocols. Four of these structures are IMPLICIT, INTEGER, DATA, and DIMENSION statements. Since the DATA statement would require 2 templates, one each for the statement itself and for the item structure in the statement, these structures would require 5 templates. More templates are necessary for arithmetic expressions. Some of these, such as the one for for incrementing the value of a variable, are syntactic units in their own right, requiring their own templates; others are built up recursively out of simpler templates. Eight more templates would be adequate to cover the ones seen in the protocols. No function calls, such as calls to SQRT or ABS built-in functions, appear in the original 4 segments though they are used in problems, PAUL and RALPH; at least one more template would be necessary for this structure. Both conjunction and disjunction are used in some of the tests in IF statements. Another 1-3 templates would be necessary to include them. Finally, the arithmetic IF statement is used in problem KEVIN; it would also require a template. Thus, at least 16 more templates, in addition to the original 18, would be necessary to model all the syntactic structures seen in the protocols.

In addition to those templates whose necessity can be demonstrated from the protocols directly, the need for other templates can be inferred from other sources. The file that the subject is given contains READ and WRITE statements and the subject uses the WRITE statement in some of his de-bugging (though he has no need for them in writing the programs proper); therefore, it may be inferred that templates for these statement types are needed. At least one template apiece will be necessary for the READ and WRITE statements themselves; possibly a second or third template may be necessary for each one if the END or ERROR options are used. The variable lists for both statements will take an additional template, since the iteration construction for I/O is not used in any other statement type. The FORMAT statement will require a first template for the statment itself, and additional templates for each format specifier. Assuming that the subject knows about E,F,I,X,A, and H specifiers, the repetition construction, the next record construction, and the quoted literal construction, a reasonable number of templates would be 9. The total needed for I/O statements would then be 13.

In addition to I/O statements, the programmer's qualifications as an experienced FORTRAN programmer may be used to infer his knowledge of other constructions. He certainly knows about CALL, FUNCTION, and SUBROUTINE statements as well as STOP and END. He also knows about COMMON and REAL declarations. These would add 7 more templates.

Adding all these figures gives a grand total of 54. This is by no means a precise figure and is no doubt an under-estimate, since it was obtained through enumeration only of known cases. Some of the structures estimated as requiring one template may take several, and there is no way to guess about templates needed for arithmetic constructions which don't appear in the protocols. It is reasonable, however, to assume that 150 is an upper bound on the total number of templates necessary to represent this subject's knowledge of FORTRAN syntax and structure. In the light of the previous discussion, this is a very reasonable figure and useful evidence that templates are a plausible model for how the programmer's knowledge about the syntax of a language is represented.

An additional perspective on this figure can be obtained from analogous figures for other tasks of comparable complexity. One candidate seems to be the number of patterns that has been found necessary to simulate the behavior of a chess master at a position reconstruction tasks (Simon & Gilmartin, 1973). At first glance, the figure calculated there, 31,000, does not seem at all comparable, since it differes by more than two orders of magnitude. Before asserting, however, that the two tasks differ markedly in this respect, it is important to ask whether the two numbers measure analogous things. In the chess task, the patterns are built up out of primitive objects, the moves of the pieces themselves. In the coding task, on the other hand, the templates are building blocks out of which more complex code structures are built. Thus, the 150 and 31,000 figures measure entirely different kinds of knowledge units.

To derive a more comparable unit, the number of basic units for chess must be calculated. The basic unit used by the Simon and Gilmartin program is the single piece, excluding pawns, giving 6 as the number of units. The use of the single piece was presumably selected on a priori grounds. In fact, it is possible that, for experienced players, the basic unit may be configurations of up to 2 or 3 pieces. If this is so, the total number of building blocks might be close to the figure for templates in coding.

6

Implications of the Model

6.1 Production Systems as Behavioral Representations

In Chapter 3, the choice of production systems as control structures was defended on the grounds of their general suitability for representing human behavior and without regard to whether they were an appropriate choice for representing programming behavior in particular. At this point it is worthwhile to ask whether some other structure might not have been more suitable. The primary characteristic of a production system is that at any given point, the choice of the next piece of behavior is made in parallel from among all the possible alternatives. This is not to say, of course, that there are no sequential dependencies in the behavior produced by a production system, but, rather, that the dependencies are a result of the specific model, not of the control structure itself. A structure that was essentially different from a production system would select behavior via a sequence of decisions that was an inherent part of the control mechanism structure. To argue that production systems are a particularly good choice for representing coding behavior requires that there is some aspect of the coding process which cannot be easily represented in a serial fashion.

In this case, a strong argument for essential parallelism can be made from the retrieval of knowledge about the association between plan elements and code. As was argued in the previous chapter, an experienced programmer has a large body of knowledge about how to code particular plan elements. Since this body is so large, serial processes in searching it ought to reveal themselves by extremely long retrieval times for infrequently used plan elements, perhaps on the order of several minutes (as versus the tenth-second retrieval time for LTM cited previously). Additionally, the protocols ought to contain some evidence of sequential elimination of unwanted information until the correct information is found. In this set of protocols, once the subject has a plan, coding of it begins almost immediately, and, while the subject may consider altenative methods of coding a plan, there is no evidence in the protocols of rejection of inappropriate, unwanted information. This suggests that the search for coding information for a plan element is made in a parallel manner. If this is so, then a production system is a particularly appropriate control structure for representing coding behavior.

6.2 Implications for the Use of Back-tracking

One of the most common ways to organize a problem solving system is as a back-tracking sub-goaler. Systems with this organization attempt to solve problems by reducing them to a set of sub-problems, the solution to which implies the solution to the intial problem. When the system fails as a sub-problem which it has attempted to solve, it returns or back-tracks to some prior, successful state. Systems of this type vary considerably along such dimensions as the strategy used to generate and select sub-goals, the amount and kind of information retained from failures, and the point to which return from occurs (Nilsson, 1971; Newell and Simon, 1972).

The model presented here is not organized as a back-tracking sub-goaler; however, it is still of interest to inquire what role back-tracking plays in programming behavior. In its purest form, back-tracking in programming would consist of completely abandoning some piece of code by erasing it or crossing it out and beginning again at an earlier point, up to which the code was known to be correct. In fact, this occurred only once, in problem DAVID. A far more common occurrence, one that has already been described in detail, is that the erroneous code is modified by inserting a line here, crossing out a variable in another place, and changing an expression in a third. This sort of behavior takes place in 31 of the 42 coding segments covering 20 of the 23 protocols. A specific example occurs in lines 36-37 of problem RICHARD in which the initialization for NEXTODD is altered; note that if backtracking behavior had taken place in this case, it would have appeared as crossing out or abandoning all the code that has been written so far and completely starting over beginning with the new initialization for NEXTODD.

As explicated in Chapver 4, the model produces a corresponding behavior using the production system and the CODE modification functions; when a mis-match occurs between a desired effect and an actual one, productions, sensitive to the particular mis-match, use the CODE modification functions to perform the necessary operations. Thus, the behavior of both the model and the subject on encountering erroneous code is to attempt to save as much of the written code as possible.

Contrast this behavior with the potential code generation behavior of systems such as GPS (Ernst & Newell, 1969) and PLANNER (1972) which, when failure occurs, abandon the entire attempt, send back a failure signal, and select a new subgoal. Since these systems are always more or less starting afresh after every difficulty, the code they produce would probably be quite clean and easy to follow while the code produced by the subject and model may be a maze of patches and "hacks." In problem JOHN, for example, a back-tracking system could have produced a solution using only two DO loops instead of the four used by the subject.

Additionally, backtracking systems can often function using only a few general principles of program construction applied repeatedly in various possible combinations. To create code in the same way that the subject and model do, on the other hand, requires a great deal of specific information on how to best correct specific difficulties. Counterbalancing these disadvantages is the main advantage of the "patch and move forward" approach used by the subject; if the requisite information is available and if a sloppy solution can be tolerated, it usually requires much less effort to produce a solution than does backtracking.

6.3 Implications for A Theory of debugging

In the first chapter the choice of program writing rather than debugging behavior for the focus of this theory was defended on the ground that a theory of program writing would have strong implications for a theory of debugging and ought to be developed first. As an example of the sort of connection that may be made, it is worthwhile to explore briefly some of the implications of this theory of program writing for error analyses.

The exploration begins with a basic premise: Errors in programming are not random or accidental in origin but are a lawful product of the structure of the problem solving system, and, if conditions at the time of the error are repeated, the error will be repeated. (i) Given this premise and this theory of programming, errors in programming fall into one of 3 classes:

1. Errors made in the Understanding process. This would include errors attributable to mis-reading or mis-interpreting problem directions.

2. Errors made in Planning. These might be further classified into errors caused by plan elements which failed to meet necessary pre-conditions, errors

----

(i) A possible exception: motor errors in typing.

due to plan elements which produce a wrong result for the overall aim of the problem, etc.

3. Errors in Coding. Further classifications in this catagory would include:

a. Errors made because the wrong code was generated for a plan element.

b. Errors made because the wrong effect is assigned to a piece of code so that it falsely appears to match the plan element.

c. Errors made because of incorrect retrieval of a variable name or expression that has been used previously in the program.

d. Syntactically incorrect code.

To use this error classification scheme as the basis for a debugging scheme, it would first have to be refined to a set of behavioral criteria. Then, these criteria would have to be applied to the protocols to obtain the actual instances of errors for use in testing the theory. Even this sort of preliminary analysis is, of course, a task requiring a sizeable amount of effort. For this reason, the inquiry here will be restricted to asking whether the mechanisms of the model are adequate to reproducing the various types of error behavior.

Since the model makes no claims about the understanding or planning processes, it is not relevant to errors occuring in them. Of the four types of coding errors, the first two are likely to have behavioral consequences of the type described in the previous section and can be reproduced by the production system. Errors involving incorrect name or expression retrieval can, as was mentioned previously, be produced by the way information is retrieved by MEANINGS. Finally, faulty templates or faulty instantiation of a template by the production system would produce syntactically incorrect code. Thus, the model is adequate for producing errors actually seen in human coding and would be useful starting point for a debugging theory based on how the bugs had occurred in the first place.

Aside from its relationship to the model, it is interesting to compare even this preliminary classification with the error classification schemes that have been used in other studies (Youngs, 1970; Gould and Drongowski, 1972). Excluding syntax errors, these schemes classify errors by statement type or function; for example, as assignment, iteration, declaration, or flow-of-control errors. Such a classification has two main drawbacks: First, it tends to be very dependent on a particular language family; the distinction between flow-of-control is obscured in SNOBOL because alternatives in pattern matches in assignment statements may directly change the flow of control. Second, it gives no basis for determining why the error occurred, and, therefore, gives only the scantiest clues as to how the error could be prevented. Schemas implemented within the outline given here, on the other hand, could be largely language independent and would provide information that could be used to reduce errors both on an individual basis and in general programming practice.

6.4 Implications for Teaching Programming

According to this theory, programming knowledge may be divided into three types:

1. Knowledge about plans and their approriateness for given problems.
2. Knowledge about the syntax and structure of specific programming languages.
3. Knowledge about how to code a given plan element.

Planning knowledge is partly dependent on Information about specific domains and is probably acquired almost exclusively by experience in programming in these domains as well as from formal study of algorithms of the sort advocated by Knuth (1969). Knowledge about language syntax is usually acquired from formal study of manuals and similar materials which give the grammatical rules for the language. Knowledge of the third type consists of information of the sort, "to perform this operation in this language, use this structure." An example for FORTRAN of this kind of information is "to test each element of a vector, write a DO loop and use the index of the loop as the vector subscript." In all likelihood, this knowledge is acquired by direct experience with using these structures and operations in writing programs.

Using these knowledge types and the respective ways in which they are acquired, certain implications for methods of teaching programming can be derived. One of these concerns an approach to program writing called "structured programming" in which a formal, proof methodology is used to insure the generation of correct programs (Dijkstra, 1972). Wirth(1973) has written an introduction to programming based on this approach. While probably not intended for use by those who are totally naive to programming, it is still of interest to inquire whether this approach would be worthwhile at the beginning level. The Wirth book can be characterized as linking a set of effects stated in formal terms to each statement type of a programming language. Relatively little attention is devoted to presenting useful higher-level constructions of these statements. Thus, students taught using only this text would probably excel in the second type of knowledge, but would be largely lacking the first type. Given a program, they would probably be able to prove its correctness, but they would have greater difficulty than students trained by other methods in constructing programs given problem descriptions.

6.5 Implications for Automatic Programming

Automatic programming in the current sense may be defined as a process by which a statement of the desired properties of a program is translated into actual code without the need for detailed specification by the user of the desired flow of control. Such systems usually also employ some semantic model of programming knowledge and make inferences from this model. Buchanan (1974) presents a more detailed discussion of the definition of automatic programming and an overview of existing systems.

There is nothing inherent in this definition that constrains these systems to have the same cognitive organization or to operate in the same way that human programmers do; however, since the human programmer is still far and away the best programming "system" known, the way humans program provides a useful prospective from which to view these other systems. Under the assumption that newer systems will attempt to incorporate what has been learned from the construction of previous ones, two more recent efforts, Buchanan (1974) and Balzer et. al. (1973), have been selected for inspection.

The Balzer system is unique in the automatic programming field in being based on a system for acquiring its knowledge about the programming domain directly from interaction with users; in other systems, this knowledge must be built into the system in the form of premises or theoroms. The knowledge thus acquired is organized as a Loose Model which is then transformed via a variety of problem solving methods into a Precise Model. The Precise Model is stated as a program in a high level programming language. When construction of the Precise Model is complete, this program can actually be run and is the final output of the system.

The Buchanan system is a more conventional one in that it has no knowledge of the application domain but must rely on the system user to state the problem in a domain independent form. An interesting feature of this system is its use of the Logic of Programs (Floyd, 1967; Hoare, 1972) as a logical basis for program generation. The logic of programs views programs as triples of the form, I(P)O, which is interpreted as "if I is true, and the program P is run, then O will be true afterward." Constructing a program is then equivalent to proving a theorom using the input-output assertions for the basic constructs of the language as axioms.

The domain acquistiion part of the Balzer system resembles quite closely the Understanding process in the theory presented here. The way in which it generates code, however, is entirely different and might be considered equivalent to constraining a human programmer to using plans which were statements in a programming language. The Buchanan system on the other hand, resembles the code process of the theory, taking plans as input and producing code as output. The Os in the I(P)O formalism are analagous to the effects in the human programming model with the difference that they tend to be stated in terms of the logical or mathematical properties of the program rather than in terms of distinctions important to the real-world actions the program must perform.

The primary drawback to the Balzer system would probably be that, since the language and structure of plans is extremely constrained, the system will probably spend considerable effort in generating them. Since the system returns to the user for more information when it gets "stuck," this may mean considerable effort on the user's part. Additionally, since the high level language used is a very specialized one, considerable work on compilers and translators may be necessary to allow the system to produce code in other languages. The drawback to the Buchanan system is that input to it must already be in a well-specified, domain-independent form, and, as Buchanan points out, this may be nearly as large a task as writing the program in the first place. Given these drawbacks, combining the two systems offers intrigueing possibilities. The domain acquisition portion of the Balzer model could eliminate the input problem of the Buchanan system. The Balzer system, on the other hand, could beneficially use the Buchanan system to permit a looser, more general plan language which, in turn, would lead to more efficient, more general problem solving. The end product would be quite like the model for human programming presented here and would be a closer approach to the flexibility and power of the human programmer.

6

Bibliography

Bobrow, R. J., Burton, R., & Lewis, D. **UCI Lisp Manual**. Information and Computer Sciences Department. University of California at Irvine. 1973.

Balzer, R., Greenfield, N., Kay, M., Mann, W., Ryder, W., Wilczynski, D., and Zobrist, A. Domain-Independent Automatic Programming. USC/Information Sciences Institute. 1973.

Boehm, B.W. Software and It's Impact: A Quantitative Assessment. RAND Corporation.1972.

Buchanan, J. A Study in Automatic Programming. Department of Computer Science, Carnegie-Mellon University. Pittsburgh, Pa. 1974.

Dijkstra, E. Notes on Structured Programming. In Dahl, O.J., Dijkstra, E. & Hoare, C.A.R. **Structured Programming** Academic Press. 1972.

Chase, W.G. & Simon, H.A. Perception in Chess. **Cognitive Psychology** 1974.

Eastman, C. CognitiveProcess and ill-defined problems: a case study from design. **Proc. International Joint Conference on Artificial Intelligence**. 1969.

Ernst, G.W. & Newell, A. **GPS: A Case Study in Generality and Problem Solving** Academic Press. New York. 1969.

Fikes, R.E. & Nilsson, N.J. STRIPS: A new approach to the application of theorem proving to problem solving. **Artificial Intelligence**. Vol. 2 1971.

Floyd, R.W. Assigning Meanings to Programs. Proc. of Symposium in Applied Mathematics, Vol. 19, 1967.

Freeman, P. & Newell, A. A model for functional reasoning in design. **Proc. International Joint Conference on Artificial Intelligence**. 1971.

Grant, E.E. & Sackman, H. An exploratory investigation of programmer performance under on-line and off-line conditions. **IEEE Transactions on Human Factors in Electronics.** HFE-8 1967.

Hayes, J.R. & Simon, H.A. Understanding writter problem instructions In Gregg, L. W., **Knowledge and Cognition.** Lawrence Erlbaum Associates, Washington, D.C. in press

Hewitt, C. Description and Theoretical Analysis of Planner. Unpublished doctoral dissertation. Massachusetts Institute of Technology, 1971.

Hoare, C.A.R. An axiomatic basis for computer programming. Comm. ACM, 12, 10, 1969.

Gelernter, H., Hansen, J.R. & Loveland, D.W. Empirical exploration of the geometry theorom proving machine. **Proceedings of the 1960 Western Joint Computer Conference,** 143-147 1960.

Gold, M.M. Time-sharing and batch-processing: an experimental comparison of their values in a problem-solving situation. **Communications of the A.C.M.** 12 249-259 1969.

Gould, J. Some psychological evidence on how people debug computer programs. **International Journal of Man-machine Studies.** in press.

Gould, J. & Drongowski, P. A Controlled Psychologcial Study of Computer Program Debugging. **IBM Research Report** RC4083. 1972

Klahr, D. A production system for counting, subitizing, and adding. In Chase, W.G. (Ed.), **Visual Information Processing.** Academic Press. 1973.

Klahr, D. & Wallace, J.C. The role of quantification operators in the **development** of quantity. **Cog. Psych. 4** 301-327. 1973

Knuth, Donald E. **The Art of Computer Programming: Volume 1. Fundamental Algorithms.** Addison-Wesley Publishing Company. 1969.

-- **The Art of Computer Programming: Volume 3. Sorting and Searching.** Addison-Wesley Publishing Company. 1973.

Moran, T.P. The Symbolic Imagery Hypothesis: A Production System Model. Department of Computer Science. Carnegie-Mellon University. 1973.

Miller, G.A. The magical number seven plus or minus two. **Psychological Review** 63:81-97. 1956.

Newell, A. Notes on the psychology of programming. Computer Science Department. Carnegie-Mellon University. (forthcoming)

Newell, A. PSG Manual. Computer Science Department. Carnegie-Mellon University. 1974.

Newell, A. A theoretical exploration of mechanisms for coding the stimulus. In Melton, A.W. & Martin, E. (Eds.) **Coding Processes in Human Memory**. Winston. 1973.

Newell, A. & Simon, H.A. **Human Problem Solving**. Prentice-Hall. New York. 1972.

Nilsson, N.J. **Problem Solving Methods in Artificial Intelligence.** McGraw-Hill. New York. 1971.

Quam, L. & Diffie, W. **Stanford Lisp 1.6 Manual**. Stanford Artificial Intelligence Laboratory. Stanford University. 1974.

Rubey, R. J. A Comparative Evaluation of PL/1. **Datamation**. December, 1968.

Schatzoff, M., Tsao, R. & Wiig, R. An experimental comparison of time sharing and batch processing. **Communications of the ACM**. v.10(5) 1967.

Sime, M.E., Green, T.R.G., and Guest, D.J. Psychological evaluation of two condtional constructions used in computer languages. **International Journal of Man-Machine Studies**. 5(1) 105-113 1973.

Simon, H.A. The Heuristic Compiler. In Simon, H.A. & Siklossy, L. (Eds.), **Representation and Meaning**. Prentice-Hall. Englewood Cliffs, N.J. 1972.

Simon, H.a. & Kotovsky, K. Human acquisition of concepts for sequential patterns. **Psychological Review**, 70:534-536 1963.

Simon, H.A. & Gilmartin, K. A simulation of memory for chess positions. **Cognitive Psychology**. 1974.

Smith, L.B. A comparison of batch processing and instant turnaround. **Communications of the ACM**. v.10(6) 1967.

Sussman, G. A computational model of skill acquisition. Doctoral dissertation. Massachusetts Institute of Technology. 1973.

Sussman, G.J. & McDermott, D.V. Why Conniving is better than Planning. Proc. FJCC 41. 1972.

Weinberg, G. **Psychology of Programming**. Van Nostrand Reinhold Co. New York. 1971.

Weissman, Larry Psychological complexity of computer programs: An initial experiment. Computer Systems Research Group. University of Toronto, 1973.

Weinberg, Gerald & Schulman, Edward Goals and performance in computer programming. **Human Factors** Vol. 16(1) 1974.

Wirth, Niklaus **Systemic Programming: An Introduction**. Prentice- Hall. Englewood Cliffs, N.J. 1973.

Young, Richard Children's seriation behavior: a production system analysis. Unpublished doctoral dissertation. Psychology Dept. Carnegie-Mellon University. 1973.

Youngs, Edward Error-proneness in programming. Unpublished doctoral dissertation. University of North Caroline at Chapel Hill